Updated 1 December 1997

FORTH GUIDE

INTRODUCTION

There are many ways to provide an introduction to any subject and FORTH is no exception. Skinner and Holland developed a system based on operand behavior. Their techniques have been successfully applied to many subjects.

Another scheme is to develop a careful step by step approach. Take one step at a time and build slowly. Always provide review of material and connection from one step to the next.

Alas, that is not the way we learned to do most things. That is not the way we learned to talk or think. Rather, we were exposed to many different areas of knowledge through experience from our early pre-school days. Bit by bit we took a little here and a little there and added it to our organization of what we already knew.

Learning in this manner is dependent upon the individual, not upon any course design. As we learned our native language we gradually expanded our understanding of the words we knew and added to our vocabulary. It is an active process for the student. There is substitute for that active process.

In this GUIDE, it is assumed that the user has a computer. That he knows how to turn it on. That he knows how to load his operating system. That he knows how to format his disks. That he has some sort of feeling for what the object code is that makes any program run. (It is a file of some sort which can be brought into the memory of his system to make it do certain things.) It is assumed that the user has used his computer for simple word processing.

It is assumed that he has learned, hopefully not from bitter experience, to make backups. He will learn how to use MVP-FORTH to make back up copies of his FORTH SCREENS disks later in this GUIDE.

Further, it is assumed that the user knows the meaning of bits, bytes, and words. It is assumed that user knows what mnemonics means. It is assumed that he has at least a vague understanding of what an assembler and compiler are. In essence, it is assumed that he has at least a beginning level of computer literacy!

It would be a big help if he also knew how to type.

MVP-FORTH includes a working vocabulary of about 140 words, labels for computer functions, with another 100 words which will be rarely if ever used by the programmer. In learning a foreign language, you can make flash cards and learn 10 or 20 new words a day. There should be no problem learning the 140 words. The problem in learning FORTH is that the functions of the FORTH words, labels, are not always what you might expect. The syntax of the FORTH language is completely foreign to many westerners. There is no way to relate FORTH syntax to that of western Indo-European languages.

The only way to learn FORTH is to let the concepts associated with the FORTH become a working part of your programming life.

This leads to a problem with any printed discussion of FORTH. How should you distinguish between the words which could be called ideograms, of the FORTH language and common English words. For example, a "," is a FORTH word. Also spaces are significant in FORTH. Proportional spacing with variable space widths makes recognition difficult. The FORTH words are not always easy to recognize when set in proportional type. Occasionally formatting instructions to the typesetter may include FORTH ideograms. Hopefully, all such occasions have been found and the formatting instructions overridden.

With this as an explanation, you are cautioned to remember that the text is a mixture of two different languages. There is no simple style which solves the problem completely. In some cases the pronounced name of a FORTH word is used within single quotes. The pronounced name is different from the symbols used to make up FORTH ideograms as noted in ALL ABOUT FORTH.

The FORTH GUIDE should ease the way for you to become aquainted with Forth. You are encouraged to use the GUIDE interactively on a system running MVP-FORTH. You would do well to have ALL ABOUT FORTH and to use it as a companion to the FORTH GUIDE. You should also have available the MVP-FORTH SOURCE LISTINGS, which is included on this disk and as Volume 2 of the MVP Forth Series. The information in those two volumes will not be repeated.

ALL ABOUT FORTH includes the 80x86 implementation of all of the words used in MVP-FORTH except the EDITOR and ASSEMBLER vocabularies. With each entry is an example. For ease of reference, all entries are in ASCII order. That does not make for easy reading.

The FORTH GUIDE is just a guide. If you have need of reference to a particular function, you should not be using this GUIDE. Rather, you should refer to ALL ABOUT FORTH. This GUIDE is divided up into sections. A collection of words which function together is discussed in each section. The interactive play among the words and their use is gradually developed.

The use of arithmetic operations is minimized. Most of you will have had an elementary school introduction to numbers. Little more than that limited expertise is needed here. Many FORTH functions require that you tell the system the necessary values before you execute the function.

How this is done is really not that important. After you are at home with MVP-FORTH and your system, you can explore the additional arithmetic functions available.

Do not try too much at one time. Reread as necessary. You will probably not understand everything the first time through. You do not need to understand all parts referred to in this GUIDE to become moderately proficient with FORTH.

An elementary principle of all programming is: KEEP IT SIMPLE, STUPID. (KISS)

Finally, this GUIDE will not tell you what applications you want to program. If you have nothing to program, you have no business learning any programming language. If you have nothing to say you have no business saying anything.

CHAPTER I

GETTING STARTED

The programming language, FORTH, harnesses the hardware to a simple group of functions. It is in a sense, the Assembler mnemonics for an ideal processor. The major use of a computer is for text or data input and output. The older machines were used for number crunching but today that use is limited on personal computers. A complete number package with floating point and transendental functions is a separate application available from Mountain View Press.

First you must have a running system with MVP-FORTH to load. Several object modules are distributed on this disk. (See the documentation file.)

If you have your printer connected and wish to log a hard copy of your session, enter a control-P. To stop the logging, enter another control-P. This DOS function is also implemented in MVP-FORTH. The FORTH function toggles a flag. When the flag is set, the screen is echoed on the printer. A hard copy is particularly useful when starting. You have a record of what you did when something unexpected happens. It can use alot of paper.

You cannot damage your hardware with FORTH. However, in moving rapidly and often without thinking, you can easily crash the system software. No problem! Just as when you stall your car, simply restart it. Often you will never know exactly what you did. Usually, a software restart of the system will be enough, but sometimes it may be necessary to turn the power off and on again. It is like car. You have to turn the ignition off then back on before you can rerun the starter.

For your feeling of security, you can leave FORTH any time.

```
BYE   <cr>
```

Note that the command must be entered in upper case. MVP-FORTH is case sensitive. When entered in lower case you will be answered with the message: NOT RECOGNIZED. What is not recognized is indicated as part of the message. Try leaving FORTH. That wasn't so hard, was it?

When FORTH is first loaded, you will see on the display a header and a version number. Type a carriage return

```
<cr>
```

The response on the screen will be "OK". The program is ready for you to enter any command you wish. The OK means that the system has finished what ever it previously was doing.

What functions can you use? There are about 140 common functions already defined and you can add to these as you wish by defining your own functions interactively.

Type a series of carriage returns and you will see a column of OKs down the side of the screen. Yes, go on and try it. This is intended to be interactive. Finally, you are at the bottom of the screen and you may notice that the OKs scroll off the top.

After an OK, type:

```
JUNK   <cr>
```

and the system responds:

```
NOT RECOGNIZED
```

Obviously, the system did not understand you. JUNK has not been defined. You have done no damage to the system. It just cannot do anything.

Well, what can you do? You will have to learn the functions which are available. To start with you will have about 140 functions to learn. You have already learned one, BYE. Try some more.

```
page   <cr>
```

The response again is:

```
NOT RECOGNIZED
```

The problem is that by convention most FORTH words are in upper case. There is a long history to this. In olden times, most terminals could only produce upper case. For portability including old terminals, only upper case has been used. There is no reason that you cannot use lower case for your own definitions if you wish. But to get started, just set the capital lock key. Now try again.

```
PAGE   <cr>
```

Note the screen is cleared and you will see the OK at the top left. It is not that difficult. If your system only produces a carriage return, check with your documentation to learn how to make it perform its correct function.

Next suppose that you want to move down the page 4 rows before entering anything:

```
CR CR CR CR   <cr>
```

And the OK appears down 4 blank rows.

Every FORTH word must be separated by one or more spaces or a carriage return. The carriage return will also tell FORTH to interpret and execute all of the words entered up to that point.

Any other combination of characters and symbols may be used in making new words. Use a mnemonic group of characters when you later define your own functions. Sometimes a long English word will have the best mnemonic value. In this case we might use CRS as a mnemonic for plural carriage returns.

Thus it would be much easier to simply enter:

```
4 CRS   <cr>
```

But this gives the message:

```
NOT RECOGNIZED
```

Now try a big step. No apology is made for this jump ahead at this point. If you do not understand, all you need to do is copy the example. Understanding will come later.

Define a function CRS which will move the cursor down any given number of rows. This is a function which will repeat the single CR function a given

number of times.

```
:  CRS  0 DO CR LOOP ;   <cr>
```

Now try:

```
4 CRS   <cr>
```

This time the program understood you. You have added a function to the system. You will note that the added function can be used immediately. There is no edit, compile, load and run. This truly is interactive programming.

What did you do to define the function `CRS`? You will see more details of the process later, but you have enough here to use. The type of function is called a 'colon' definition. The definition begins with a colon! It is an interactive compiler directive.

Next you entered your name for the new function. The name can be any string of up to 31 alphanumeric and symbol characters except a space and a carriage return. The only reason the name you choose has any mnemonic value is that you selected it. You could just as well have typed:

```
:  !@#$%^&*  0 DO CR LOOP ;   <cr>
```

Try that and prove it to yourself. Then test it:

```
4  !@#$%^&*   <cr>
```

Next in the colon definition you entered a zero, 0 . The value is the beginning range for the `DO ... LOOP` structure. The way you will use the function is to first enter the number of rows you wish to skip. The value you entered will be one more than the other end of the range. The range 0 through 3 has four values - the desired number.

Inside the `DO ... LOOP` structure place the already defined functions you wish. In this case you placed `CR` within the loop. You wanted the single function, `CR`, to be repeated.

Finally, end the colon definition with a semicolon. As soon as you type the carriage return, the new definition is ready for use.

Again, what happens if you make a mistake entering the definition? First of all no real damage is done. You will be given a short message along with the location of the error. The definition will not be added to the dictionary of the system.

You cannot use a `DO ... LOOP` structure outside of a colon definition. If you try, you will have the system mark the `DO` and give you the message `COMPILE ONLY`. Another carriage return will return the usual OK prompt.

Some times the system will appear to go to sleep. Without restarting your program, a simple sequence of carriage returns often will get you back to the OK prompt. Who knows where the system went?

There is one exception to this rule - the message: NOT UNIQUE. This is not really an error. It means that the name you chose had been previously used for something else.

All previous uses of the word will remain unchanged, but all subsequent uses will have the new function.

HELLO

Now try to clear the screen and display the message HELLO, in the middle of the monitor. This will introduce the function of SPACES which is similar your new CRS , but has already been defined.

```
PAGE 12 CRS 37 SPACES  ." HELLO"  12 CRS   <cr>
```

You have just entered interactive instructions which will be interpreted and execute as soon as you enter the carriage return. You also used the `." ... "` pair of instructions. Whatever appears between the `."` and a terminating `"` will be printed on the terminal.

The steps of our interactive instructions are as follows: Clear the display and home the cursor, move down 12 rows and in 37 spaces, type HELLO, and move down 12 more rows to take the OK out of the way. Once you become familiar with FORTH, you will find that the instructions are just as clear in FORTH.

Finally, make a function which will execute these instructions with a single command. This is adding a new colon definition to your FORTH.

```
:  HELLO  PAGE 12 CRS 37 SPACES
   ." HELLO" 12 CRS ;     <cr>
```

You can break up your definition into a couple of lines if necessary. A carriage return inside a colon definition is just compiled not executed. Now test your new definition.

```
HELLO   <cr>
```

With these new tools, you can place any message you wish anywhere on the screen. You could have used "Hello world". Make up your own exercises. Choose names which are meaningful to you. When you are all done, you will have added several words to the vocabulary of functions available in your implementation of FORTH.

Perhaps you do not remember all of the words which you have added. You can see the new words as well as all of the old ones with the command:

```
VLIST   <cr>
```

Hit any key to stop the scrolling. Then hit any key again to continue scrolling. While the scrolling is stopped, you can hit any key twice in rapid succession to leave the scrolling. Several MVP-FORTH functions which scroll data on the screen operate in a similar manner. In faster machines the delay may be too short. You can learn to fix that later by

refering to ALL ABOUT FORTH.

The display begins with the latest definition first. Perhaps you will have started a definition which did not get finished. Even though the function is not available because you perhaps made an error in defining it, you will see the word in the dictionary. This is a peculiarity of the system.

Eventually, you will want to learn the function of the words displayed with `VLIST`. But as you have seen, there are things you can do in FORTH with only a small selected portion of them. Also many of the words are system primitives which you will probably never need. MVP-FORTH is completely open and documented. You are free to change it any way you wish. First you really should learn what is included and always make back up copies. Please do not distribute your versions as MVP-FORTH.

As you add words you are gradually filling up the dictionary and the space available in memory. It is best to have only those words in the dictionary which you will have occasion to use in any given application. The implementation of MVP-FORTH includes a number of common functions and serves as a basis for developing your own system. You might want to eliminate some of the words. Perhaps you will have no use for the `ASSEMBLER`. You can make the choice better after you have had some experience.

For now, each time you start the program, you will have to enter your favorite definitions again. At least at this stage that is the case. There are some things you will learn to do later. You will be able to write functions and save the source for your later use in FORTH screens. You will be able to load these screen without retyping them. Still later, you will learn to save the current object code image as a new file as the new application. For the present you will have to start over each time which is no big deal especially if your remember: KEEP IT SIMPLE!

CHAPTER II

FORTH SCREENS

If you have started with one of the standard implementations of MVP-FORTH, you will have an option available under the FORTH function, `CONFIGURE`. Try it.

```
CONFIGURE   <cr>
```

You will be shown the number of drives available and a density code which corresponds to the density of the disk you have on each of the

drives. You will note that FORTH starts with drive 0 and increases. For two drives the designations will be `DR0` and `DR1`. Hopefully, these designations will be properly set for your system. If so, simply respond with a carriage return. The function will abort unchanged.

The designation of the drives beginning with 0 is the way the system actually identifies each drive. If your system uses drive A , the A is converted to a 0 for access to the drive. Why not just call it drive 0? Your system may have different densities than in the options give. Later we will see how to modify that.

Return to your operating system with `BYE`.

```
BYE   <cr>
```

You will need a special disk for use with FORTH. It is called a FORTH SCREENS disk. This disk will have no system files on it. It can only be used from FORTH. This is why the distribution FORTH SCREENS disk cannot be read by your system. You cannot use your system COPY command but DISKCOPY will work. Later you will learn how to duplicate the distribution FORTH SCREENS disk with FORTH. For now you will not need it.

Note: In the Documentation for this disk you will see how to use SCREENS in DOS files and even use text files as a source for MVP-FORTH applications.

Use the operating system on your computer to format a new disk. Label that disk FORTH SCREENS.

You can use your newly formatted FORTH SCREENS disk in any drive. For the present, load FORTH from a copy of your MVP-FORTH system disk in drive 0 (drive A). Then remove the SYSTEM disk and replace it with your newly made FORTH SCREENS disk. At this point, it is necessary for the System Disk and the Screens disk to be in the same format. See if the disk part of the system is working under FORTH.

```
75 LIST   <cr>
```

You should see a screen full of a single character or symbol. When the disk was formatted, a value was placed in each byte on the disk. That byte will print as some symbol or perhaps a lower case e . So far so good. Now see if you can clear the screen of those symbols.

```
WIPE   <cr>
 75 LIST   <cr>
```

You should now see a blank FORTH screen. At the top is indicated the screen number, 75 . Then 16 blank lines numbered 0 through 15 should appear. Many FORTH programs are written on such screens.

A number of editors are available to write on FORTH screens. You do not have to learn a whole new editor at this point. The MVP-FORTH kernel

has a very rudimentary editor. It is not a full screen editor. It consists of a single command. Now try it.

```
0 PP ( THIS IS LINE 0 )   <cr>
75 LIST    <cr>
```

Everything following the single space after PP will be placed on line 0. The PP is a mnemonic for put except P has already been used.

If all is well, you should see the text following the PP on line 0. If not you might try again. Should that fail, try reading the documentation supplied with your program. In a worse case Mountain View Press will support you.

Now save the screen to disk.

```
FLUSH   <cr>
```

The drive should activate and the screen will be written to disk. Check to be sure.

```
75 LIST   <cr>
```

The disk drive should again activate and you will see the screen with what you had written on line 0.

Try writing other material on other lines.

```
FLUSH   <cr>
```

The screen will be saved back to the disk. Without FLUSHing the screen to disk, it only remains in a RAM buffer. Now reLIST it. Remember you will need to tell LIST which screen to list by entering the desired screen number first.

```
75 LIST   <cr>
```

If you forget to enter the screen number you will get an error with SCR marked and the message EMPTY STACK. The function needs a value which you did not give it. Try again.

This rudimentary editor is very useful. You can enter any keystroke which is not possible with some other editors. However, there is no way to correct part of a line with it. You must type the line over. If you are a good typist you can do that as fast as you can move the cursor. It will encourage you to keep the lines short. Later you will learn about the line EDITOR available in MVP-FORTH, but this is not the time to learn another editor. Stay with the rudimentary one for now. Ultimately, you may want to add one of the screen editors available for your system.

Each line on a screen contains room for 64 characters. You do not need to use them all. In the example we placed the comment on line 0 within parentheses. It is a custom to place a name for each screen on line 0. Often it is convenient to include your initials and a date. This will help you keep

track of what is on each screen.

```
70 80 INDEX   <cr>
```

This will list line 0 from each of the screens 70 through 80. It makes a sort of table of contents. You will note that all screens except for 75 contain only the format character. Line 0 of screen 75 has its title.

Make the title you choose, meaningful to you. Perhaps you can list the functions you have defined on that screen.

The comment is placed in parentheses so that it will be treated as a comment when you later load, compile, the screen. Screen 75 has been chosen for example because most disk systems have at least 75 screens. But you can try any screen number you wish. You can check the limit of screens on your disk by doing `INDEX` through a large number.

```
75 999 INDEX   <cr>
```

You can interrupt the scrolling by hitting any key. Hitting any key again will restart the scrolling. To leave the scrolling, first stop it with a single key, and then hit any key twice in rapid succession. You may have to adjust the delay for the speed of your computer.

The system will list the 0 line of each screen up to the limit on the disk and then stop. `INDEX` will not go on to the next disk. If, however, you try to list a screen number larger than that on your 0 drive, the system will automatically move on to `DR1`. You will notice that your second drive activates. It will be better for the present to stay within the range of screens on `DR0`. You will have to learn to be responsible. There is no safety check on this feature.

MVP-FORTH presumes the programmer is responsible. It will not limit your access to the system and will happily destroy everything if you wish, or even if you only enter the wrong information.

It is often good to use one screen as a directory to the location of material on the other screens on a disk. An early screen number is good for this. On the other hand you do not want to use the first few screens. Those on the zero track of a disk often have some formatting information about the disk. It is best not to write over this information. Therefore you might make screen 10 a directory.

```
10 LIST    <cr>
WIPE    <cr>
0 PP ( DIRECTORY )    <cr>
1 PP 75 A TEST SCREEN   <cr>
FLUSH    <cr>
10 LIST    <cr>
```

You could now define a word to show you your directory.

```
: DIR 10 LIST ; <cr>
DIR  <cr>
```

There is no need to indicate a carriage return at the end of each line. By now that should become a reflex for you. You will also have learned by this time that each FORTH function is separated by a space.

Now return to screen 75 and enter the function CRS which you defined earlier.

```
75 LIST
WIPE
0 PP ( CRS )
2 PP : CRS ( n --- )
3 PP    0
4 PP    DO CR
5 PP    LOOP ;
FLUSH
75 LIST
```

Your definitions should be neatly laid out on a screen. Several conventions have been used but they are only conventions. You are free to adopt them or not as you see fit. But you and others will probably find the screens easier to understand later if you adopt some clear conventions.

The placement of the name of the function being defined on line 0 within parentheses as a comment has also been mentioned. Place it within parentheses so that it will not be loaded later. Especially during development, but even later extra spaces and white space can be used to advantage to set things off.

Line 1 is left blank as a matter of style. It serves to set off the beginning of a definition on the next line.

On line 2, the colon definition is started in the first column. When the screen is loaded later this line will perform exactly the same function as when it was entered interactively.

Following `CRS`, the name for multiple carriage returns, you have another comment within parentheses. This is not necessary but later it will serve to remind you what you did. The "n" indicates that a number must be entered before the name of the function. The parentheses indicate that this information is just a comment and is not necessary for the definition.

The placement of a 0 on line 3 all by itself seems like a waste of space. But it is necessary if we are going to start the `DO` structure on the beginning of a line. Again, the 0 is the beginning of the range of iterations for the structure. The number of iterations will have been entered before the command.

Start structures as `DO` on the beginning of a line. It makes them easy to spot. You will note that the entries on all lines within a definition have been indented 3 spaces. The spaces make the definition easier to read. Then enter the functions to be performed within the `DO ... LOOP`. Each

DO must be paired with a LOOP. It is easy to see if this is done if the LOOP is entered with the same indent as the DO.

Finally, end a colon definition with a semicolon.

In order not to lose what you have done, it is a good idea to FLUSH a new screen back to disk immediately. You can then LIST it again and see what you have done. At this point, since the lines are so short, it is very easy to retype any line using PP which appears incorrect. Then FLUSH the corrected version.

Now, you can test the screen.

```
75 LOAD
```

LOAD compiles what is on the selected screen just as if you had entered it interactively at the terminal. Writing a definition on a screen and loading it is almost as interactive as writing it at the terminal. The time to load a screen is usually only a fraction of a second. Though this is not strictly interactive programming, it is certainly much faster than the old way of editing, compiling, loading and running. When you have made a mistake or wish to change you program, it is quicker to edit the screen and then load it than it is to interactively type in the definition again.

The response will be the familiar OK --- that is, if all is well. If not you will get an error message showing the screen number and line number where the error occurred. Then you can reLIST the screen, use the rudimentary editor to make the necessary corrections, FLUSH and LOAD the screen again.

Next debug the routine. Perhaps the definition of the function you wrote would LOAD, but will it do what you wanted?

Now you can repeat a series of commands you used earlier:

```
PAGE 12 CRS 37 SPACES ." HELLO" 12 CRS
```

The old program should come out the same.

Try writing the program as a colon definition on screen 76. Follow the example above. Put each function on a separate line. Perhaps you can include the number on the line with CRS and SPACES. The ." HELLO" can be on a separate line. Gradually, you will learn to think in FORTH.

```
76 LIST
WIPE
0 PP ( HELLO )
2 PP : HELLO
3 PP    PAGE
4 PP    12 CRS
5 PP    37 SPACES
6 PP    ." HELLO"
7 PP    12 CRS
```

```
8 PP ;
FLUSH
76 LIST
```

If what you entered is correct, you can LOAD it:

```
76 LOAD
```

This lesson is a little long, but there are a few more related functions to learn. If you wanted to have several programs which would put different messages in the middle of the display you could just modify the screen 76. But you might want to keep that screen too.

```
76 77 COPY FLUSH
77 LIST
```

A copy of screen 76 is now present on screen 77. You can now edit this screen by changing the name of the function and the message. FLUSH that one too.

The copy function is trivial. The buffer containing screen 76 is simply renumbered 77. FLUSH then writes the newly numbered screen to the disk.

Next time you load FORTH from your copy of the MVP-FORTH system disk, be sure to replace it with your FORTH SCREENS disk before proceeding. Then you can reload your programs.

```
75 77 THRU
```

These relatively simple FORTH commands and techniques should serve as examples for you to expand upon. An exact functional definition of each of the resident commands can be found in ALL ABOUT FORTH. You have saved your new definitions on your FORTH SCREENS disk.

CHAPTER III

INSIDE MVP-FORTH

You have been given a quick tour of some of the things which you can do with MVP-FORTH. It is time to take a break and review some of those things.

You were introduced to about 15 of the functions which you have available in MVP- FORTH. This is already about 10% of those which you will have to master. Without any explanation you were plunged into 'colon' definitions of new functions. You were shown how a structure can be used to repeat functions.

Occasionally, the terms dictionary and word were used. No explanation

was given. Hopefully you tried the examples and learned something about how to use the language. Now you will learn a little more about how it is put together.

Each defined function is given a name. The name can be any combination of characters except space and carriage return. Any group of symbols can be used as a name. Such names are hardly words as we think of them in English.

Perhaps you have heard of the Chinese use of symbols. Their written language is composed of characters known as ideograms. They have no immediate relation to their pronunciation. Each ideogram encompasses a concept of some sort. The concepts are assembled in lists which provide meaning through associations.

In a way the names for the FORTH functions are similar to the Chinese ideograms. Only in those cases in which the names look like English words could one call them words. In the other cases the names might better be called ideograms.

But still, the names of FORTH functions are often called words. In ordinary writing about FORTH they are referred to as words as in this book. But do try to understand that the use of English-like words does not imply English-like syntax. This is simply not the case. Nor is there syntax as it is used in many other programming languages.

The ideograms, or words, are collected in a linked list within the computer's memory. The list is very much like a common dictionary. Each entry in a common dictionary includes some information about the word: its pronunciation, its part of speech, its origin, and then information about what the word means.

Each entry in the FORTH dictionary in your computer includes some information of a similar type: the name length, a pointer to another word, a pointer which tells the system how to interpret the meaning which follows. All of this information is encoded in a series of bytes in computer memory.

Every time you call a function by entering its name, the location of the name is found in the dictionary. The pointer which tells the system how to interpret the following meaning is used to actually execute the function. All of this may seem a little vague, but if you take the time to examine the entries in the FORTH dictionary, you can read the encoded information. You will find that the analogy is not too bad. Details are described in ALL ABOUT FORTH.

Now there is no need to master this sort of information, but you might just want to take a look. For this purpose, FORTH has a `DUMP` function which looks very similar to the one you

probably have in the collection of programs which came with your computer. DEBUG is such a program.

You might want to take a look at your computer memory with this MVP-FORTH function. By entering first the beginning address followed by the number of bytes you wish to inspect and then the word `DUMP`, you will see such a display. For this to have any meaning you must know how to interpret the hexadecimal numbers used to represent the information in memory. The appearance is similar to that in your system program.

To the right of the display, you can see the character or symbol which each of the byte codes will produce. Not all bytes represent printable symbols. Those which do not are represented by a period.

Perhaps you will want to skip ahead a little, but again you might want to take a quick look. Dump the object code for FORTH.

```
256 20000 DUMP
```

The numbers you entered were decimal. `DUMP` translates this to hexadecimal in the display. You can again stop the display any time by hitting any key and continue again by hitting any key again. Remember, you can escape from scrolling by first stopping it and then hitting any key twice.

As the image of memory scrolls by, you will occasionally see words which you recognize. All of the entries in the dictionary will appear if you scroll through the entire dictionary. Later you will look more closely at the form of a dictionary entry.

For now you will see that the dictionary starts at low memory and extends toward high memory. Any new function you define, such as the `CRS` which you did earlier, will be added on top of the dictionary. So you can say that the dictionary grows up in memory.

High memory is used for special functions in FORTH. These functions include the buffers for the screens. Other buffers are also located in high memory. These buffers are transient and do not have to be saved with the object code for your FORTH. They are reconstructed every time you start the program.

Try the word `LIMIT`:

```
LIMIT .
```

The system will print a number. The function of the constant, `LIMIT`, is to store the highest memory location to be used when you start the program. The "." is also a FORTH word. Its function is to print the number you have entered. It is pronounced 'dot' and means print. The symbol, "." , is an ideogram. Neither its pronunciation nor its function is obvious, hence, ideogram.

You might find that LIMIT has a value around 24000. If your system has 64K of memory, you could make your FORTH highest memory location much larger. But you will have to know something about the use of the rest of the memory in your system before you change the value of LIMIT. For now, the 24K value of LIMIT will allow you to do much of what is included in this guide. Later you will learn to change the value and restart the FORTH program. Some implementations on this disk have already made the change.

Now that you have learned to use the DUMP functions, you might find it interesting to look at a buffer containing a screen. First find a screen with something on it. Maybe one of the earlier screens is available on your FORTH SCREENS disk.

```
75 LIST
```

If it is not, or you cannot find a screen with something entered on it, go back and write something on a screen and LIST that.

Now try a DUMP on the memory buffer containing that screen.

```
75 BLOCK 1024 DUMP
```

Using the function BLOCK instead of LIST will tell the system the beginning address of the desired buffer in memory. You must then tell the system the number of bytes you wish to inspect. The full screen of 16 lines each containing 64 characters has 1024 bytes.

You will note that many of the bytes are encoded with the hexadecimal value of 20. That is the code for a space. Most of the screen is filled with blanks. On the right side of the display you will see the text which is present on the screen. Remember, you can scroll through this listing in the same manner as before.

You will probably find the display given by LIST much more useful than that given by DUMP. But the information contained in both is exactly the same. LIST formats the data in a block as text while DUMP can display the actual contents of each byte within a block.

Earlier we learned to LOAD a screen such as:

```
75 LOAD
```

Occasionally you will find an error with a blank space marked. How can a blank be an error? A LIST of the screen will display nothing wrong! In such a case try;

```
75 BLOCK 1024 DUMP
```

Look for a "." in the symbol display at the right. Any non-printing character will be displayed as a period. Perhaps you inadvertently entered a control character. FORTH will try to interpret the control characters even if you do not see them displayed with LIST.

At this point you might want to inspect the bytes on a freshly formatted disk. Find a screen which is filled with the single symbol produced by the formatting byte. Now do a DUMP on that screen buffer. You will see a field with a single byte value. Perhaps it will be all E5's.

Next, WIPE the screen and then DUMP it instead of LISTing it. This time you will see a field of all 20's. Explore your system. You will become more at home with what is going on inside of that 'little black box'.

FORTH is no different from any other program in your computer. The information is stored in a series of bytes. Any program, your word processor, FORTH, or whatever, converts those bytes to a convenient format. An advantage of FORTH is that, should you be interested in how the data looks in memory you can easily see it interactively with DUMP.

FORTH also has the feature that you can change any byte in memory. For that matter, you can change any byte on your disk. The ability to perform these changes provides the power of FORTH. But with the power comes the danger. You, the programmer, are left responsible for changing or not changing the contents of byte locations in memory.

With the power of FORTH, you can, at any time, inspect any resident part of your operating system. You can in fact change any part of your operating system. You can even access anything on your system disk as well as your FORTH SCREENS disk. With this power, you can easily contaminate your system disk. More than with any other language, you must develop the habit of making backup copies of all of your work. You can never tell when an unrecognized typo will contaminate either your running program or your disk and with it perhaps some other program on the disk.

One interesting experiment is to examine the directory of a system disk. Make a copy of a system disk so that you will not lose anything important. Then try listing screens beginning with screen 1. Usually you will see garbage displayed on your monitor. This is simply because any machine language in your computer memory is not organized in text to be formatted by LIST.

Provided your system does not crash by trying to print an escape sequence to your monitor which changes its function, the garbage will not hurt anything. You might even occasionally hear a beep. Whenever the system tries to display a hexadecimal value 07, the system will beep. At least this is true on many systems.

```
7 EMIT
```

Soon, on one of the early screens you will see scattered on the screen, the names of some of the files on your system disk. Now, instead of LISTing that screen, try dumping it.

```
3 BLOCK 1024 DUMP
```

If you, directory is located on screen 3, you will see it displayed out on the right side of the screen. Actually, you will have displayed all of the information stored with the entry in the directory on the disk. This is what is known as the file control block, (FCB).

In this lesson you have been introduced to a relatively few new FORTH words. But perhaps you will have been given a little more of a feel for what is going on inside your system. Later we will use some of these tools to explore more of the details of MVP-FORTH.

CHAPTER IV

PARAMETERS

You will have noticed that some of the FORTH functions require some parameters. The value of the parameters given to the system is converted by FORTH to 16-bit binary values for storage. They are always used as 16-bit binary values.

As MVP-FORTH starts, the system is prepared to accept numerical values in decimal. For many purposes decimal notation is most satisfactory. The number system we learned in school is decimal based. We do have just ten fingers.

But as you have already seen in the output of `DUMP`, hexadecimal notation is more convenient for some purposes. It is a matter of visualizing the values as represented in memory. There is no group of bits which can be completely described with a single decimal digit. The binary, octal and hexadecimal number systems allow this.

The problem is to change the number base used. MVP-FORTH provides a simple solution. All numerical entries are converted to binary for storage in memory. The value of the current base is stored in the word BASE. By simply changing the value stored in the word `BASE`, the input and output base or radix can be changed.

To change to the hexadecimal system, simply enter `HEX`.

```
HEX
```

In this mode, hexadecimal digits 0 through F can be entered. You can then convert to the decimal mode:

```
DECIMAL
```

You have already used a period, 'dot', to cause a number to be printed.

Now you can switch back and forth between number bases.

```
HEX 0F DECIMAL .
```

The system will promptly print 15. You have converted the hexadecimal on input to a decimal value on output. Now try it the other way around:

```
DECIMAL 10 HEX . DECIMAL
```

This time the system will print the value A . You have converted to hexadecimal.

Working with the print command, 'dot', you will find that when you give the system a value and print it, the value is no longer there. Printing a value consumes the value. There will be times when you would like to see the value which you earlier gave without consuming it. MVP-FORTH has a command to do that: 'dot-s'.

```
.S
```

Now try some exercises. Enter some numbers and print them with both the `.S` command as well as the 'dot' command. You will note that if you have not entered a number and try either command you will be shown the message STACK EMPTY. You will also note that you can repeat the `.S` command as many times as you wish and always have the same result. However, with the 'dot' command, once you have printed all of the values previously entered, you will get the message STACK EMPTY.

Now try putting in several values:

```
1 2 3 . . .
```

You will note that the last number entered will be the first one printed, and so on until all of the values are printed. The value farthest to the left will be a 3 followed by a 2 followed by a 1. Now try the `.S` command:

```
1 2 3 .S
```

The values are printed in the reverse order from that in which you entered them. This is just like executing a series of 'dot' commands. Now some people find that they would prefer to see the numbers in the order in which they were entered rather than the reverse order. There are two MVP-FORTH commands which control the function of the `.S` command.

They are `.SL` and `.SR` . These commands change a flag in a word .SS . You can easily select the mode of representation you find most natural.

While you are entering values and printing them, you might explore the way the simple arithmetical operations function. Like many Indo-European languages, the operator is at the end. The operator must know what it is going to operate on before it can operate. Rather than entering a value then the operation which must be set aside to wait for the second value before executing, simply enter the values and then the operation.

Some hand held calculators do the arithmetic operations the same way.

The method is called Reverse Polish Notation or RPN. There is really nothing complicated about it. World wide, probably more people think this way than the other way. The problem is that there are a number of hand held calculators which work the reverse way. Also some of our schools teach arithmetic the reverse way. This may have something to do with the aversion many people in this country have to arithmetic.

But it is really simple enough. Just experiment with your system and see how it works. You will get a message should you do it wrong. Simply try it again.

Now suppose you know you have put in some numerical values in error and wish to erase them. First you might want to examine the current values with the `.S` command. Then you could use a series of 'dot' commands until you get the message - STACK EMPTY.

Another way to clear the values already entered is to enter a word that is not recognized. You will immediately get the answer: NOT RECOGNIZED. You can then try the `.S` command and see the message: STACK EMPTY.

Now someone will wonder if all of the numbers in all of the bases are entered in the MVP-FORTH dictionary. Certainly not! Rather, if the entered word is not found in the current MVP-FORTH dictionary, the system will attempt to convert it to a number depending on the value currently stored in the variable `BASE`. If successful it will save the binary value. If the system is unable to do either, then it will give you the message: NOT RECOGNIZED.

A collection of functions is used to implement this ability of the system. They might be called primitives. They are included in MVP-FORTH and are available for your use. But you will probably not want to work with them until much later, if at all.

The words `DECIMAL` and `HEX` are already defined in the dictionary. The names of other bases are not. They can be added easily enough with colon definitions. Later you will learn more about colon definitions. For now just use the format indicated here.

```
: OCTAL 8 BASE ! ;
: BINARY 2 BASE ! ;
```

The only new word is the exclamation point (`!`). It is pronounced 'store'. The colon definition of `OCTAL` enters the value of the new base (8) and then the location of the variable `BASE` and finally stores the one in the other. The colon definition `BINARY` is exactly the same except for the value to be used.

Now you have available the words to do any sort of number conversion you wish. You can experiment with a number of them.

```
10 HEX .S DECIMAL .S
OCTAL .
BINARY .S
DECIMAL
```

It is probably a good habit to always end up in the decimal number system. That way your habit will always tell you where you are.

Now suppose you have to examine the numbers entered a number of times. This calls for another colon definition:

```
: #CONVERT
   HEX .S
   DECIMAL .S
   OCTAL .S
   BINARY .S
   DECIMAL ;
```

Now you can execute this string of commands very simply. The name chosen is a little long but it combines the idea of a number (#) and the English word convert. Should you have a number of these operations to do, you might just for the occasion give it a name easy to type, `XX`, for example. This has no mnemonic value but is easy to use.

It is very convenient to interactively enter a colon definition in this manner just for the occasion. You can then promptly forget it.

```
FORGET #CONVERT
```

The function erases all definitions written after the word which you forget.

You now have a problem. You could gradually build up the vocabulary in your MVP- FORTH dictionary with more and more convenient utilities. You might have more than a thousand such utilities. Then you have to remember or lookup just exactly what each of those functions are.

MVP-FORTH has included only the words which have been proven most convenient and the primitives associated with them. For a major part of FORTH programming you can do very well with the included words. Once you have mastered them, you can easily write a special function using a string of characters which is easy to type, but with no mnemonic meaning and forget the function when through. This way you will always know what the function is.

Now go back and try using MVP-FORTH for simple arithmetic calculations. The common operations are `+` , `-` , `*`, and `/` . These have all of the expected functions. They operate on 16-bit integers and yield 16-bit integers. This is sufficient to cover many of the early calculations you will need to make. The numerical operations do not end here though. A number of mixed operators are available as well as double precision, 32-bit, operators. Also MVP-FORTH has a complete floating point, quad precision, and trancendental number application. You can wait until later to explore some of those operations.

Remember that the values must be entered before the operator.

```
2 2 + .
3 6 * .
6 2 / .
4 2 - .
etc
```

You can go on from here and learn as much as you care to, about the numerical operators. Explore what happens at the boundary conditions!

CHAPTER V

SYSTEM DEFINING WORDS

Consider this chapter as a model application even if you are not actually interested in the details of the system application.

You have been introduced to writing colon definitions in MVP-FORTH as well as the idea that all definitions are located in your system's memory in a "dictionary". You have been told that each entry contains a variety of information. Let us now examine that a little more closely.

Each entry has a header, and usually some data. As pointed out, the header includes a count byte indicating the length of the name, a link to other words in the dictionary, and a pointer to machine code which tells the system what to do with the data which follows.

A colon definition is unique in that the pointer to machine code tells the system to execute each of the two byte pointers which follow in a sequential list. The data in a colon definition is a simple array of pointers. The array ends with a pointer to the function, `EXIT`. This function terminates a colon definition. There are a few exceptions to this rule about which you will learn more later.

With a colon definition we can include any word already defined or any numerical value currently acceptable to the system. This is certainly the most common type of defining structure in MVP-FORTH. A colon is used to start the definition and a semicolon is used to terminate it. The colon is really a MVP-FORTH function which creates a word in the dictionary and prepares to build the array of pointers to the functions to be executed. Rather than entering the function `EXIT`, the semicolon enters that for you in just the right place.

There are several other types of defining words besides colon. You have already seen constants, `LIMIT` for example, and variables such as the user

variable BASE. The MVP-FORTH words CONSTANT and VARIABLE are also defining words. They are available for your use.

```
10 CONSTANT TEN
VARIABLE VALUE
```

Each of these lines will add a word to the MVP-FORTH dictionary. The first one has the property of giving the system a value every time it is executed. TEN will now give the value 10 without going through a number conversion. It will run much faster than a number conversion.

A VARIABLE will give not the value but a pointer to the value. The data field contains the value. Note that the content of a variable is not initialized when it is defined. Your program must initialize the value.

The advantage of having two ways of defining numerical values is a matter of convenience. Each can do much the same thing. If you are always going to have to get the value of a variable and it may change frequently you will find it easier to use VARIABLE in defining the word. If, on the other hand, the value is going to remain fixed, you will probably find that CONSTANT is better to use in defining the word.

Next you will have to learn to put values in variables and to get values from the variables. MVP-FORTH speaks of these functions as 'store' and 'fetch' respectively. The notation is cryptic. An exclamation point (!) is used for 'store' and the at symbol (@) is use for 'fetch'. The 'store' function is particularly powerful and thus particularly dangerous.

At this point you now have the power to make a mistake and destroy your system. You will have to learn to be responsible. But even if you destroy your system, all is not lost if you have developed good work habits. Always maintain a backup copy of your work.

The worst part of making a mistake using the 'store' function is that you do not recognize that you have done it. You have inadvertently stored an unknown value in an unknown location. Not until sometime later will a problem show up. It will show up when the location you have modified is used by the system. This may occur immediately. But it may also occur hours later. In that case you will not even be thinking about a mistake you may have made in the past.

As a consequence, do not be overly disturbed if your system quits working for you or does very peculiar things. Perhaps you are not perfect and some time ago made a mistake. Rather than try to debug the system, go back to a copy which you know is good and start over again. Perhaps a quick look to see if your new routine is correct is worth while. But before spending too long be sure that it is your current new routine which is causing the trouble.

Hopefully, the point has been made: be a responsible programmer and you

can enjoy the power of FORTH. You will probably have to learn this by experience.

The functions 'store' and 'fetch' operate with 16-bit address values as well as 16-bit data values. Thus two bytes are accessed beginning at the designated byte address. There are two other functions, ' C-store' (`C!`) and 'C-fetch' (`C@`), which operate on 8-bits at the designated address. The "`C`" is mnemonic for character.

All this aside, you were learning to manipulating variables. When you give the name of a variable, the system finds a pointer to the variable. To find out what is in the variable you must 'fetch' the value.

```
VALUE @ .
```

And to display it, you will have to print it with 'dot'. Before the content of the variable `VALUE` has been initialized, you will fetch garbage. To initialize a variable or to later place a value in the locations:

```
100 VALUE !
VALUE @ .
```

You have stored the value 100 at the location pointed to by `VALUE`. You have then inspected the new value. You could have used another MVP-FORTH function to inspect a value:

```
VALUE ?
```

The question mark has been defined as 'fetch' and 'dot'. It takes one less stroke. The function has a long history in FORTH but not everyone uses it regularly. Again it is left up to the user.

At this point, try changing a value in a constant such as `LIMIT`. This exercise was avoided earlier because you should have some comfort with MVP-FORTH before using it for things like this.

MVP-FORTH has a function which will search the dictionary for the word which follows and return a pointer to the beginning of the data field if it is found. The data field is sometimes referred to as the parameter field.

```
' LIMIT @ .
```

The apostrophe by itself is called 'tic'. It will search the FORTH dictionary and return a pointer to the data field. There are several ways you can inspect the contents of a data field. If you know that the word's data field contains a numerical value you can fetch the value and print it. That is what you did above.

Now that you know how to change the value of a constant, you can increase the size of your MVP-FORTH image in memory. To do this you have to know something more about your system. You will not want

MVP-FORTH to write over some other necessary resident part of your system. You will have to refer to your system documentation to determine a safe value for LIMIT. As a sample and probably safe value for LIMIT you could use 32000.

```
32000 ' LIMIT !
```

Having changed the value of LIMIT you need to recreate the high part of MVP-FORTH in the system's memory. This is done by reconfiguring the system.

```
CHANGE
```

The function of CHANGE is to restart FORTH using the current value of LIMIT. Later you can change other properties of the system with CHANGE.

After executing CHANGE you will have lost all of your new definitions unless you take particular care to save the current status. The problem is that in low memory are kept a number of start-up values such as the location of the top of the dictionary. Before you execute CHANGE, execute FREEZE. The function of FREEZE is to save all of the current system status values in the proper location in low memory. Once this is done CHANGE will use the current status values.

You could use 'tic' to return the data field for the variable VALUE as well. But it is not necessary because it is the same thing that is returned by a variable.

```
VALUE ' VALUE . .
```

Demonstrate for yourself that this is true.

You can also use DUMP to examine the data field of any FORTH word.

```
' LIMIT 10 DUMP
```

This function not only displays the contents of the data field of LIMIT, but 16 bytes beginning with it. DUMP will always fill out a row. Now comes some possible confusion. When 16-bit values are stored in memory, many eight-bit systems store the least significant byte before the most significant byte.

If you did not modify the value in LIMIT, you will see the first byte is 00 and the next one is 60. The values are in hexadecimal and the 60 comes before the 00 making the value 6000 hexadecimal. The remainder of the data displayed with DUMP begins the header for the next word in the FORTH dictionary.

You can now change the data for the calculation used to access the disk with the 'fetch' and 'store' functions. You might need to change the data used by the system to find the selected information on the disk.

```
CONFIGURE
```

A carriage return will abort this function with no change to your system.

The information displayed in the prompts for CONFIGURE are never accessed by the system in accessing the disk. In fact not all of the information is always used. It depends upon the implementation.

The data is arranged in arrays for each of the available parameters. The arrays are blocks-per-drive (`BLK/DRV`), sectors-per- track (`SEC/TR`) and sectors-per-block (`SEC/BLK`). Now you can inspect these arrays.

```
BLK/DRV 10 DUMP
```

The array is composed of 7 16-bit values arranged in order according to the density codes 0 through 6 selected with `CONFIGURE`. That is, each value uses two bytes. Remember that many systems place the low order byte first. Also remember that the display of `DUMP` is in hexadecimal.

Examine each of the other arrays in a similar manner.

`CONFIGURE` allows you to select the number of drives your system has and to assign one of seven density codes to each drive (0 through 6). The code you select for a specific drive accesses the data from these arrays as necessary. If you have only one type of drive and drive density, you can assign the proper information to the first item in each of the arrays. Then in spite of the prompt, when density code 0 is selected those values will be used.

Perhaps this exercise has moved ahead of your experience. If your system is working and the disks are being properly accessed, you will have no need to make any changes to these arrays. However, you might have found it interesting to see how they are laid out.

How do you put the values into the arrays? To create an array you can start with the definition of a variable, then instead of a new definition, you can simply add a series of values to the dictionary.

The system maintains a dictionary pointer, `DP`. You can display the value at `DP` by the MVP-FORTH words `HERE` 'dot'.

```
HERE .
```

There are two functions which add values to the dictionary: 'comma', and 'C-comma', (, and `C,`). The comma will add a 16-bit value to the dictionary and advance the value in the dictionary pointer 2. `HERE` will then return a value two larger. The C-comma will add an 8-bit, character, value to the dictionary and increment the dictionary pointer only one address.

Thus you could make a new array of 7 values:

```
VARIABLE TEST-ARRAY
0 TEST-ARRAY ! 0 , 0 , 0 , 0 , 0 , 0 ,
```

Now inspect what you have done.

```
' TEST-ARRAY 10 DUMP
```

If all is well, you will see 14 bytes of 0. You are beginning to get the feel of using FORTH.

An alternate way of creating an array of 7 16-bit values is:

```
VARIABLE TEST-ARRAY-2 12 ALLOT
```

Execution of VARIABLE in defining the word allots 2 bytes for the value. You need to ALLOT 12 more bytes in your dictionary for the rest of the values. With this method, the values are not initialized.

Now to access an item in the array, you need to select an item number. Remember the first item will be 0. Then double the item number because each item takes two bytes, and add that to the pointer returned by the variable TEST-ARRAY. You now have a pointer to the desired item in the array, so fetch the value there.

```
4 2* TEST-ARRAY + @ .
```

This sequence will inspect the 4th 16- bit item in the array TEST-ARRAY.

If you are going to do this often, you might define an appropriate pair of words.

```
: @TEST-ARRAY ( item --- )
   2* TEST-ARRAY + @ ;

: !TEST-ARRAY ( n, item --- )
   2* TEST-ARRAY + ! ;
```

You have combined two FORTH mnemonics for the name of each function. In both cases you need to tell the system which item you are accessing.

At this point you have explored three types of defining words: colon definitions, CONSTANTs and VARIABLEs. There are other kinds of defining words and you can easily write your own, but you will have to wait for that.

You have also learned one way to create arrays and to 'fetch' and 'store' values from them.

CHAPTER VI

VECTORING

Most computer languages restrict the functions available to previously defined routines. In many applications you might find it desirable to change the functions. To use a new function, you might have to recompile your entire program.

What you would like is a way of changing a selected function and have the new function used every time the old one is referred to in the already completed part of the program. An example of this is the MVP-FORTH word PAGE.

The function of PAGE is to clear the screen and home the cursor. The code necessary to do this is usually different from one terminal to the next. If you have a generic MVP-FORTH Programmer's Kit version, you will find that PAGE does not perform as defined. Your documentation will tell you how to proceed but you will see how vectoring PAGE is a big convenience. You can write the necessary program and vector PAGE to it.

The generic MVP-FORTH Programmer's Kit version vectors PAGE to CR . The run time routine of a carriage return seems safe enough even if it does not do what you want. You must now find the escape sequence or code necessary to do the desired function.

```
27 EMIT 69 EMIT 26 EMIT 3 0 0 0 16 INTCALL DROP
```

On various systems one or another of the above routines serves the desired purpose. EMIT causes the preceding byte value to be sent to the monitor. The first line sends an escape code followed by an upper case E. The second sends a control-Z to the monitor. Finally, after setting up the necessary values, the last line does an interrupt call which is possible on some systems. It is beyond the scope of this guide to cover all of the possible codes.

You can test the possible functions as indicated above. When you find one that works, put it in the dictionary as a new colon definitions.

```
: <MY-PAGE> 27 EMIT 69 EMIT ;
```

Next you can verify your new definition. FORGET and modify it as you find necessary.

You are now ready to vector the new definition into your system. A group of variables has been assigned names combined with a preceding apostrophe. These variables contain pointers to the respective execution addresses. For PAGE there is 'PAGE. The function of PAGE is simply to get the pointer from 'PAGE and EXECUTE it.

EXECUTE is an MVP-FORTH function which will execute the word pointed to by the value previously given. That is, one can give the system the address of the pointer to the code routine for a given word and the EXECUTE it.

PAGE is defined as follows:

```
: PAGE   'PAGE @ EXECUTE ;
```

Other vectored functions work the same way.

```
' <MY-PAGE> CFA EXECUTE
```

This will be another way to test that the value you are going to put into 'PAGE is correct. Remember! In MVP-FORTH, the address returned by a 'tic' is the address of the beginning of the data field, which is sometime referred to as the parameter field address of a word. The pointer to the machine code used to interpret the data is two bytes before the data field. It is known as the code field address, (CFA). A FORTH function, CFA, decrements the data field address two bytes. This then is the address to be executed.

You can now put the code field address of <MY-PAGE> in 'PAGE.

```
' <MY-PAGE>  CFA 'PAGE !
```

Hence Forth PAGE should perform as expected.

Note: Since parenthesis is used with comments, MVP-FORTH uses angle brackets to enclose many primitives.

Vectoring can be used in a variety of other ways. Suppose that you wish to have all of the alphabetical characters input from the keyboard in upper case even if you have not set the cap-lock key. MVP-FORTH gets all input from the keyboard with the function KEY. KEY is vectored to the code field address pointed to in 'KEY. In MVP-FORTH, 'KEY points to the code field address of a run time routine, KEY. All you need to do is check the value input by KEY and if it is a lower case character, subtract a hexadecimal value of 20 from it. If it is not a lower case character, do nothing more.

You can easily build an array for translating the ASCI values from the keyboard to Dovorak values and convert your system.

Exploring this problem will allow you to examine a few more MVP-FORTH functions. Perhaps you will want to develop this program on a scratch screen so that you can modify it as you debug your function.

First you will want to know what the code values for a lower case "a" and "z" are. You could go find them in a table somewhere, but why bother. You can easily check at the keyboard. Use the function KEY before you modify it.

```
KEY  <cr>
```

The system will wait for you to enter any key at the keyboard. You can then see what that key was with a 'dot'. Use this function to determine the value of lower case "a" and "z". All codes within this range should be changed by reducing the value by a decimal 32 or hexadecimal 20.

```
: <UPPER-KEY>

   DUP 97 <   OVER 122  >
   OR 0=
   IF 32 - THEN ;
```

Several new MVP-FORTH functions have been introduced. You will learn more about them later, for now just use them. The new function first executes the original run time function to get a value. You will not want to use the word KEY, because you will eventually change its meaning. You then need to use the values you determined for lower case "a" and "z". First make a copy of the value returned by <KEY>. Then enter the code for a lower case "a" and make a logical comparison. A truth flag is left. Next save that flag and get another copy of the original value and see if it is above a lower case "z". You then have two truth flags which you combine with the logical operator OR. Finally you reverse that flag which says that the value is not out of the desired range. You can then apply the MVP-FORTH conditional structure IF ... THEN. More on this later.

You may find that FORTH is easier to read than English text.

The purpose of this exercise is to illustrate vectoring and not all of the code necessary to make all alphabetic characters upper case.

You can try your new function.

```
<UPPER-KEY>  <cr>
EMIT
```

The new function will wait for a key and then EMIT will cause the symbol for code which was left to be displayed on the monitor. When you are convinced that the new function is working correctly, you can vector it into KEY.

```
' <UPPER-KEY>  CFA 'KEY !
```

Now you no longer have to worry about having the capital lock key set.

You can use vectoring for a number of other functions. They are suggested as exercises. Suppose you wish to keep track of the number of lines you are using. You could define a variable as a line counter. When you are ready to start set it to zero. Then define NEW-CR to increment the counter before executing the run time function <CR>. Finally, vector the code field address of your NEW-CR into 'CR.

Perhaps you will then want to execute a form feed each time you have output 60 carriage returns. Now you can put some logic into your NEW-CR. If the value in the counter is 60, output a form feed and reset the counter to 0 before completing the function. You can then vector that into your system.

There are many features of MVP- FORTH which you can change without

having to completely rewrite the source code. You can add your new vectored functions.

Vectoring is particularly useful for forward referencing in developing you programs. You can temporarily vector in the code field of a NOOP function, if you have one. If not, define one. Later, when you define the actual function you can change the code field address in the pointer variable to the correct function.

If you have done any changes of the initial vectored functions you may well have a problem with FORGET. Suppose at some time you decide to FORGET all of your new definitions back to the beginning of your program. Should you have revectored a function in your program you will have forgotten the run time routine you defined. The amusing thing is that the system may not notice at first. Not until you reuse the area of memory which you freed by FORGET, will the system notice. But then you will find that you have vectored to something other than the intended function. In all likelihood the system will crash.

Vectoring is most useful in program development. Learn to use it effectively.

CHAPTER VI

VECTORING

Most computer languages restrict the functions available to previously defined routines. In many applications you might find it desirable to change the functions. To use a new function, you might have to recompile your entire program.

What you would like is a way of changing a selected function and have the new function used every time the old one is referred to in the already completed part of the program. An example of this is the MVP-FORTH word PAGE.

The function of PAGE is to clear the screen and home the cursor. The code necessary to do this is usually different from one terminal to the next. If you have a generic MVP-FORTH Programmer's Kit version, you will find that PAGE does not perform as defined. Your documentation will tell you how to proceed but you will see how vectoring PAGE is a big convenience. You can write the necessary program and vector PAGE to it.

The generic MVP-FORTH Programmer's Kit version vectors PAGE to CR .

The run time routine of a carriage return seems safe enough even if it does not do what you want. You must now find the escape sequence or code necessary to do the desired function.

```
27 EMIT 69 EMIT 26 EMIT 3 0 0 0 16 INTCALL DROP
```

On various systems one or another of the above routines serves the desired purpose. EMIT causes the preceding byte value to be sent to the monitor. The first line sends an escape code followed by an upper case E. The second sends a control-Z to the monitor. Finally, after setting up the necessary values, the last line does an interrupt call which is possible on some systems. It is beyond the scope of this guide to cover all of the possible codes.

You can test the possible functions as indicated above. When you find one that works, put it in the dictionary as a new colon definitions.

```
: <MY-PAGE> 27 EMIT 69 EMIT ;
```

Next you can verify your new definition. FORGET and modify it as you find necessary.

You are now ready to vector the new definition into your system. A group of variables has been assigned names combined with a preceding apostrophe. These variables contain pointers to the respective execution addresses. For PAGE there is 'PAGE. The function of PAGE is simply to get the pointer from 'PAGE and EXECUTE it.

EXECUTE is an MVP-FORTH function which will execute the word pointed to by the value previously given. That is, one can give the system the address of the pointer to the code routine for a given word and the EXECUTE it.

PAGE is defined as follows:

```
: PAGE   'PAGE @ EXECUTE ;
```

Other vectored functions work the same way.

```
' <MY-PAGE> CFA EXECUTE
```

This will be another way to test that the value you are going to put into 'PAGE is correct. Remember! In MVP-FORTH, the address returned by a 'tic' is the address of the beginning of the data field, which is sometime referred to as the parameter field address of a word. The pointer to the machine code used to interpret the data is two bytes before the data field. It is known as the code field address, (CFA). A FORTH function, CFA, decrements the data field address two bytes. This then is the address to be executed.

You can now put the code field address of <MY-PAGE> in 'PAGE.

```
' <MY-PAGE>  CFA 'PAGE !
```

Hence Forth PAGE should perform as expected.

Note: Since parenthesis is used with comments, MVP-FORTH uses angle brackets to enclose many primitives.

Vectoring can be used in a variety of other ways. Suppose that you wish to have all of the alphabetical characters input from the keyboard in upper case even if you have not set the cap-lock key. MVP-FORTH gets all input from the keyboard with the function KEY. KEY is vectored to the code field address pointed to in 'KEY. In MVP-FORTH, 'KEY points to the code field address of a run time routine, KEY. All you need to do is check the value input by KEY and if it is a lower case character, subtract a hexadecimal value of 20 from it. If it is not a lower case character, do nothing more.

You can easily build an array for translating the ASCI values from the keyboard to Dovorak values and convert your system.

Exploring this problem will allow you to examine a few more MVP-FORTH functions. Perhaps you will want to develop this program on a scratch screen so that you can modify it as you debug your function.

First you will want to know what the code values for a lower case "a" and "z" are. You could go find them in a table somewhere, but why bother. You can easily check at the keyboard. Use the function KEY before you modify it.

```
KEY  <cr>
```

The system will wait for you to enter any key at the keyboard. You can then see what that key was with a 'dot'. Use this function to determine the value of lower case "a" and "z". All codes within this range should be changed by reducing the value by a decimal 32 or hexadecimal 20.

```
: <UPPER-KEY>

   DUP 97 <   OVER 122  >
   OR 0=
   IF 32 - THEN ;
```

Several new MVP-FORTH functions have been introduced. You will learn more about them later, for now just use them. The new function first executes the original run time function to get a value. You will not want to use the word KEY, because you will eventually change its meaning. You then need to use the values you determined for lower case "a" and "z". First make a copy of the value returned by <KEY>. Then enter the code for a lower case "a" and make a logical comparison. A truth flag is left. Next save that flag and get another copy of the original value and see if it is above a lower case "z". You then have two truth flags which you combine with the logical operator OR. Finally you reverse that flag which says that the value is not out of the desired range. You can then apply the MVP-FORTH conditional structure IF ... THEN. More on this later.

You may find that FORTH is easier to read than English text.

The purpose of this exercise is to illustrate vectoring and not all of the code necessary to make all alphabetic characters upper case.

You can try your new function.

```
<UPPER-KEY>  <cr>
EMIT
```

The new function will wait for a key and then `EMIT` will cause the symbol for code which was left to be displayed on the monitor. When you are convinced that the new function is working correctly, you can vector it into `KEY`.

```
' <UPPER-KEY>  CFA 'KEY !
```

Now you no longer have to worry about having the capital lock key set.

You can use vectoring for a number of other functions. They are suggested as exercises. Suppose you wish to keep track of the number of lines you are using. You could define a variable as a line counter. When you are ready to start set it to zero. Then define `NEW-CR` to increment the counter before executing the run time function <CR>. Finally, vector the code field address of your `NEW-CR` into `'CR`.

Perhaps you will then want to execute a form feed each time you have output 60 carriage returns. Now you can put some logic into your `NEW-CR`. If the value in the counter is 60, output a form feed and reset the counter to 0 before completing the function. You can then vector that into your system.

There are many features of MVP- FORTH which you can change without having to completely rewrite the source code. You can add your new vectored functions.

Vectoring is particularly useful for forward referencing in developing you programs. You can temporarily vector in the code field of a `NOOP` function, if you have one. If not, define one. Later, when you define the actual function you can change the code field address in the pointer variable to the correct function.

If you have done any changes of the initial vectored functions you may well have a problem with `FORGET`. Suppose at some time you decide to `FORGET` all of your new definitions back to the beginning of your program. Should you have revectored a function in your program you will have forgotten the run time routine you defined. The amusing thing is that the system may not notice at first. Not until you reuse the area of memory which you freed by `FORGET`, will the system notice. But then you will find that you have vectored to something other than the intended function. In all likelihood the system will crash.

Vectoring is most useful in program development. Learn to use it effectively.

CHAPTER VII

STACKS

You have perhaps wondered about the parameters which have been used in some of the MVP-FORTH functions. To `LIST` a screen you must first tell the system the number of the screen to be `LIST`ed. Sometimes you have seen that a function both needs a value and leaves a value.

Many functions in most languages need parameters and leave parameters. MVP-FORTH uses a very simple scheme for handling such parameters. It is a little like using a pronoun in the English language. The pronoun refers to its antecedent noun. You always need to be careful to keep the references straight. Should you confuse the antecedent reference you will confuse your English statement.

One way to think about values given to your computer system is similar to providing a list of references. You need to keep the references straight. If you slip you may put garbage into your program.

The order of inputting the variables is important. The order of the parameters for a FORTH function is also important. They must match. If they do not, there is little checking done in FORTH and the system may get lost. The order of values input is the reverse of the order of use by the system. This is the scheme of last in first out - LIFO.

So long as you, the programmer, remember that the order required for specific functions and make sure that they are available in that order, there is no need to burden the system with specific names for each value. As you learn to work with such a system you might find that many applications are easily solved without naming each value given the system. The process of naming is time consuming and is really an extra step for you.

The system stores the values given it in the form of a stack. The data stack of MVP-FORTH, also referred to as the parameter stack, is often the stack used by the system processor. Most computer processors use a built in stack for keeping track of the return address on subroutine jumps and calls. Interrupt routines also make use of the system stack. The design of an efficient stack for a processor is most important. MVP-FORTH usually makes use of the underlying system stack for its data stack.

In this manner data values can be stored in the system. The system stack pointer is placed within the area of memory used by MVP-FORTH. The location is setup in the high memory of FORTH when the program is loaded. The directives are stored in low memory.

Which way the stack moves depends upon where you stand. If you see the low memory as the bottom of FORTH and high memory at the top of FORTH then you will find that most processors build the stack as stalactites rather than stalagmites. That is the stack actually grows down.

On the other hand the analogy of cafeteria trays as a stack with the stack growing up has often been used. Which ever way you conceive of the stack you will refer to the top of the stack. Actually, the stack does not really move. All that moves is the value in a pointer to the 'top of the stack'. When a value is given the system, it is stored in the location pointed to and the location is incremented. All elements in the usual stack are 16-bit and thus take two address locations. The pointer is incremented or decremented by 2.

The stack then is an array of bytes. The beginning of the stack is the reference, and the pointer moves to successively lower addresses as values are added to the stack and increased as values are removed from the stack. The program automatically handles the pointer. But in MVP-FORTH, you have access to the origin of the stack. The initial value of the stack pointer is stored in the user variable (a special type of variable) `SP0`. The function, `S0`, fetches the value of `SP0` and leaves it available for use.

`SP0` is often referred to as the 'bottom of the stack'. Be careful with the image you have of the function of the stack. The bottom of the stack is the highest memory address possible in the stack.

In MVP-FORTH, the data stack moves down from high memory toward the top of the dictionary. The size of the dictionary is limited by the number of values to be used on the stack. Likewise the size of the stack is limited by the size of the dictionary.

You can inspect the stack any time you wish. To prevent changing the stack, your definition cannot add to the stack or subtract from it. You need to select a value far enough below the initial stack pointer to include all possible values on the stack.

```
: DUMP-STACK
   S0 128 - 128 DUMP ;
```

This function will dump a large segment of memory below the bottom of the stack and display it for you. Enter a sequence of digits which you can recognize and then `DUMP-STACK` and find them. Now try printing some of the definitions with a series of 'dot' commands and then try `DUMP-STACK` again.

You now see what is actually happening with the stack when you enter things. You will notice that the DUMP function makes extensive use of the stack. To make efficient use of the stack as a temporary store of values without specific names and to pass them to FORTH functions, it is well to visualize this structure. But if all of this is too much for you, it is not really necessary.

There are a few more FORTH functions which help in understanding the stack. You can always find the number of items on the stack with the function DEPTH. The value returned is the number of items on the stack before the number is returned. You will see a problem DUMPing the stack while working on the stack. You can also find the value currently being pointed to by the stack pointer, (SP@). This is defined as a code routine because the stack pointer of the system is used. It is a system register which must be moved to the stack.

The .S function uses some of these more primitive functions in a non-destructive display of the contents of the stack. It is much easier than using DUMP-STACK which you defined above.

There will be times when the order on the stack is not the order you want for a particular function. Or perhaps you will want to preserve a value on the stack which the function will consume. Or for any one of a number of other reasons, you will want to manipulate the stack.

You have already used a few of these functions. DUP is perhaps one of the most common stack function. You will sometimes see the functions which operate on the stack called stack operators. This does just what it says: it duplicates the value on the top of the stack. You can satisfy yourself of the functions by executing them one at a time and using the .S function before and after.

```
10 .S DUP .S
```

If you want to display just the top value of the stack and still have a copy of it left, you must duplicate it first. Remember the 'dot' consumes the value stored on the top of the stack.

DROP is a function which disposes the value on the top of the stack without displaying it. SWAP exchanges the top two items on the stack. OVER moves a copy of the second item on the stack to the top. This adds an item to the stack. ROT moves, not copies, the third item on the stack to the top and slides the original top two down to fill the space formally occupied by the item moved to the top.

Usually, these few functions will do almost everything you will want to do to the stack. But there are times you will want to do a few other things. You could, of course, define a function to suit your needs. Some of these are available. PICK requires the number of the item you wish to copy to

the top of the stack and replaces the selection number with the value found at that location. ROLL also requires the number of the item you wish to move to the top of the stack and slides all of the rest of the values down. If you need to use these two functions very often you might want to rethink the solution to your problem. Othertimes it is easier to just go on and use them for a quick dirty solution.

MVP-FORTH is really a two stack machine. A second stack is usually implemented in FORTH. It behaves exactly as the system stack, but is not as efficient. It is called the return stack. The return stack is used by FORTH to keep track of the addresses to be executed next.

Only within a colon definition can you move values from the top of the return stack to the top of the data stack and back. There are occasions when this is particularly useful. It saves defining special variables for temporary use. But you must understand what you are doing and see that you finish the colon definition with the return stack in exactly the condition you found it.

The FORTH function which moves the top of the data stack to the top of the return stack is >R, ('to-R'). The one to move the top of the return stack to the top of the data stack is named R>, ('R-from"). One of the advantages of using the top of the return stack for a temporary storage is that you can also copy the return stack back to the data stack with the FORTH function R@ ('R-fetch).

The full exploration of the potential of these latter MVP-FORTH functions is probably best done by a study of the code using them. They are included here to call to your attention the fact that they exist.

One other data stack function which is convenient in conditional structures is ?DUP. This function does a DUP only on the condition that the value on the top of the stack is not 0. This saves a small amount of programming if you want to DROP the value if it is 0. This function is often used in conjunction with the conditional IF ... THEN structure to be discussed later.

Few exercises have been given in the use of these stack functions. You should make up your own exercises.

```
1 2 3 4 5 .S
```

This will place some numbers on the stack and show you what you have to start with. Then try each of the functions and reexamine the stack.

```
DUP .S
. .S
DUP .S
DROP .S
SWAP .S
```

```
SWAP .S
OVER .S
DROP .S
ROT .S
ROT .S
ROT .S
?DUP .S
DROP .S
0 .S
?DUP .S
```

If you have trouble getting started, these will get you started. If at any time you are unsure of a function, you can interactively set up a similar exercise and test the function.

One of the problems with programs which do not run is that you have added some bug in the handling of the stack. There are several approaches to debugging such a program. One is to carefully adopt some format in which you can write out the stack contents after the execution of each FORTH function. In complicated programs this is probably the most efficient way to begin.

As you begin to think in FORTH, it will become almost second nature to use the stacks as it is to use pronouns in the English language. But even then, you will have occasional bugs which are difficult to find.

In such cases, you can scatter a number of `.S` functions through your definition. When the function is then executed each `.S` will display a copy of the stack. Occasionally, it is useful to echo the output on the printer so that you can carefully follow the progress.

The data stack is where FORTH temporarily stores values you or the program give the system. It saves the need of many temporary labels. It often clarifies the actual functioning of the program. The stack serves to pass parameters to successive functions. It is somewhat different from the manner of passing parameters in many programs. Where needed, it is possible to add more formal definition of parameters.

CHAPTER VIII

DEFINING WORDS REVISITED

Earlier we discussed the three basic defining words: 'colon', `CONSTANT`, and `VARIABLE`. MVP-FORTH includes two others: user variables and code definitions. The latter will be discussed with the assembler. The space for user-variables was fixed in the source for the system and is not easily

changed.

User variables, return the address containing the named variable. To this extent it appears exactly like any other variable. However, the location is defined in the MVP-FORTH dictionary as an offset from the base address of all of the user variables. This base address is set up in high memory when FORTH is loaded. Initial values are moved from low memory to high memory on start up.

The more interesting capability of MVP-FORTH is the ability to add your own defining words. In essence, this gives you the power of interactively adding to the FORTH compiler. There are few other languages with this power. It means that you can define your own data types and structures as needed for your application. You are not forced into using only those data types and structures provided by the native language. It also means that you do not have to include any more data types and structure than those required for your application.

There are three steps to the process of adding and using new defining words. You must write the new defining word, you must use the new defining word to define new words and finally you can execute the new words.

In each step, FORTH functions are executed. Therefore, each step has a run-time function. The first two steps each compiles the definition for the next step.

Step one is usually a colon definition whose function is to compile the new defining word. This is equivalent to adding to the FORTH compiler interactively. It defines the structure of the header and the contents of the data fields, and in a second part defines the ultimate function to be assigned by step two to the run time in step three.

Step two uses the defining word compiled by step one to compile a new word in the FORTH dictionary and assigns the new function to be executed by that word. This step is on a par with other defining words as `CONSTANT` and `VARIABLE`.

Step three runs the function of the newly defined word.

Step one usually uses two primitives in the colon definition of the new defining word.

They are `CREATE` and `DOES>`. Though they are usually used in colon definitions, `CREATE` may be used interactively.

`CREATE` creates a header structure for a new entry into the MVP-FORTH dictionary from the word which follows it. When the new word defined with `CREATE` is executed, the address of the beginning of the data field is

left on the stack.

The DOES> part begins the definition of a function which will be executed each time a member of the family of new words is executed. The code field of the newly defined word as set by CREATE is modified to point to a subroutine which will execute the sequence of high level FORTH words which follow DOES> in step one.

The combination of these two words produces one of the most powerful functions available in MVP-FORTH. You should take the time to study the use of these words.

First examine CREATE alone.

```
CREATE XXX
```

When CREATE is executed interactively, it compiles a header for a new word,

XXX, in the FORTH dictionary. No space is allotted to the data field. Examine the memory image of XXX with DUMP.

```
' XXX HEX 10 - 20 DUMP
```

First find the data field address even though no data has been given. In this case that should be the same address as returned by HERE. Then go to hexadecimal. This is desirable because the DUMP is in hexadecimal with 10 hex bytes on each line. By going back 10 hex bytes from the data field address, you can establish a fixed location for the header structure.

The header structure will always appear right justified on the first line of the dump. When this is the case, the last two bytes will be combined as the code field. The code field contains an address which points to machine code to be executed. In this example that machine code will return the address of the data field. You will eventually learn to recognize the pointer in the code field for the 'colon', CONSTANT and VARIABLE definitions. FORTH words defined with the assembler will have the contents of the code field pointing directly to the data field which in fact contains the machine code to be run.

The two bytes preceding the code field contain a pointer to the name field of the word defined immediately before the new one. These two bytes are called the link field. They provide the link from the present word to the rest of the words in the dictionary.

Since you have defined a word with three characters, the three bytes preceding the link field will contain the character code for the name. The last byte will not look the same. It has the highest order bit set too. The fourth byte before the link field in this case will have the value 84. The first, or lowest order five bits of this byte contain the number of characters used in the name. The highest bit is set to flag the beginning of the name

field. The other two bits are used for special purposes. They will be discussed later. This fourth byte before the link field is the beginning of the name field.

Each of the fields begins at an address. The code field address points to two bytes which contain the address of the machine code. The link field address contains two bytes which point to the name field. The name field address contains the beginning count byte followed by the characters in the name. It is a variable length field.

It will probably help for you to explore the rest of the FORTH dictionary in a similar manner. This is important enough to the understanding of all of FORTH, to take a short detour at this time. Try the following:

```
' ?CONFIGURE HEX 10 - 100 DUMP
```

This is a long definition. You will see that the name field begins near the beginning of the line. On the right hand side of the dump you will see ?CONFIGURE. The final E of the name ?CONFIGURE will have the 8th bit set and will not print.

You can see how all of the prompts for CONFIGURE are embedded in the data field of this word. You could use this DUMP to find the prompts which you might wish to change. Determine the code for the desired characters and stick them in with C!. This is the way the various implementations on the disk have beeen modified.

To return to your new word, XXX, you have not used any bytes in the data field. Suppose you wished to make the new word behave as a variable. You could compile a value in the dictionary in the location of the data field. The definition would then be:

```
CREATE XXX 0 ,
VARIABLE YYY 0 YYY !
```

Now examine each of these new words. You will see that the pointer in the code field of both is the same and the data is the same. You have found two ways of defining the same function.

The contents of the code field points to a piece of assembly code which can be found in the MVP-FORTH assembly source listing as a part of the definition of CREATE. There it has been given the assembler label DOVAR. The name is not in the FORTH dictionary but you might want to add it as a constant:

```
' XXX CFA @ CONSTANT DOVAR
```

Now you can examine the contents of the code field address of a constant. In the MVP-FORTH assembly source listing this has been given the label DOCON. You can add to your FORTH dictionary a constant with this name.

```
' DOVAR CFA @ CONSTANT DOCON
```

While you are examining the contents of code fields, you might take a look at that for a colon definition. In the MVP-FORTH assembly source listing the machine code segment has been given the label DOCOL. You can also add this address to your FORTH dictionary as a constant.

```
: NOOP ;
   ' NOOP CFA @ CONSTANT DOCOL
```

Defining the word NOOP was the quickest way to have a word which you are sure is a colon definition. If you remembered one, you could have used it as well. You might find it interesting to see what these code sequences look like in the assembly source code also on this disk. They are discussed in ALL ABOUT FORTH.

The MVP-FORTH defining word USER is used to define user variables. These are located in space allocated in high memory when the program is loaded. No extra space is available in this table. Therefore, you will have no occasion to use this word to add more user variables to your dictionary. As a matter of information, you will discover in the MVP- FORTH assembly source listing that the assembly code segment for this word has the label DOUSE. The defining word creates a constant which is then interpreted with this code segment.

Now return to step one in which a defining word is defined.

The other half of the definition of a defining word, determines what it is you want each of the new family of words to do. Perhaps this statement is a little confusing with defining of defining. Try to keep in mind the steps. In time the process will become clearer. This is one of the more difficult concepts in FORTH.

The second part of step one is compiled by the function of DOES>. The high level FORTH words entered after DOES> will be compiled like any other colon definition. They must terminate with a semicolon.

Try another approach to understanding the CREATE ... DOES> structure of defining words. Examine it in connection with a word you already know - CONSTANT. When defining a defining word in stage one such as CONSTANT, the run time routine defines CONSTANT.

```
: CONSTANT
   CREATE ( n --- )
   ,
   DOES> ( --- n ) @
;
```

The above is the definition of a defining word which will perform exactly as the already defined CONSTANT. The run time routine makes a colon definition which has two parts: 1/ The creation of a new header in the FORTH dictionary and 2/ The storage of the function which any new word subsequently defined with this word will execute, which in this case is

simply to fetch the value there.

The redefinition of CONSTANT is an example of stage one. When you use CONSTANT to define another word you are at stage two of the process. When you use the defining word CONSTANT, you are executing the run time function of the defining word.

```
10 CONSTANT TEN
```

In stage two the word TEN is compiled into the FORTH dictionary. In the code field of TEN is a pointer to machine code which will start execution of the FORTH words compiled after DOES> in stage one. In this example the function is 'fetch'.

Finally, in step three the word TEN is executed. The execution starts with the address of the data field of TEN and executes the words compiled after DOES> in stage one. In the example, the single function 'fetch' is executed and the value of ten is placed on the data stack.

```
TEN .S
```

A number of other permutations are possible. In the create portion you can poke into the data field with comma and C-comma, any values you wish. You can also execute any other FORTH functions you may wish. Suppose you want to keep track of the number of new words you are adding with your new defining word. You could define a COUNTER and initialize that to 0. Then, every time you use the defining word, increment the counter:

```
CREATE COUNTER 0 ,
: NEW-CONSTANT ( n --- )
    CREATE 1 COUNTER +!
    , DOES> @
;
```

Another FORTH function would be to create a table containing the execution addresses of all new word definitions. You could then execute any new word by fetching the contents of the item number in the table and executing it. This technique has been used in developing language vocabulary drills.

Consider another application. Set up your FORTH dictionary as a foreign language dictionary. This requires a careful understanding of the structure and function of MVP-FORTH.

What you desire is a defining word which will allow you to add two words to the FORTH dictionary. The function of each word will be to display the other word. Stage one would begin as follows:

```
: ADD1
   CREATE
   DOES>
   ID. 10 SPACES
;
```

The first part of our definition defines a word which will always display

the next word in the FORTH dictionary. This is the function of `ID.` which is already defined in MVP-FORTH. The `10 SPACES` is to keep the OK from being appended directly to the displayed name.

```
: ADD2
   CREATE
   DOES>
      LFA @ ID. 10 SPACES
;
```

The second part of our definition defines a word which will always display the previous word defined in the FORTH dictionary. Note that in both cases nothing is placed in the data field of either word by the process of adding them to the dictionary.

```
: ADD ADD1 ADD2 ;
```

When the words are combined, the data field of the first word becomes the name field of the second word. At run time, entering the first of a pair of words will take the data field address as returned from the `DOES>` and display the name field at that location. At run time, entering the second of a pair of words will take the pointer to the data field returned by the `DOES>` and move to the link field and thence to the name field of the preceding word and display that.

The new defining word, `ADD`, is used in stage two as follows:

```
ADD ONE EIN
```

You can use `ADD` to add as many pairs of words to your FORTH dictionary as you wish. You will later learn how to use vocabularies in FORTH. You could then set up separate vocabularies for French, German, Spanish, etc. You might want to try that as an exercise.

Then in stage three you can execute any of your newly defined words.

```
EIN
ONE
```

The amazing feature of the foreign language feature, is the minimal amount of space required to add these new data structures to your FORTH system. You also have the power to modify your compiler to accomplish the desired result. Given the same problem, it would be interesting to see how you might use any other computer programming language to accomplish the same result.

Use `DUMP` to examine the FORTH dictionary around `ONE` and `EIN`. Add more word pairs to your dictionary and examine them.

These are just a few illustrations of the power of defining words in MVP-FORTH. You are limited only by your ability to formulate the solution to your problem. Often that means that you are limited only by your ability to understand your problem.

CHAPTER IX

DOUBLE NUMBERS / 2 NUMBERS

MVP-FORTH provides for both 16-bit and 32-bit number functions. The 16-bit numbers take only 2 bytes of memory while the 32-bit numbers take 4 bytes of memory. In those same 4 bytes required for a 32-bit number you could just as easily have 2 16-bit numbers.

The same data space can be used in either of two ways. This is the same thing as having two separate data types taking up similar space. Other programming languages make extensive use of data typing. This is not done in MVP-FORTH. You are free to use the space any way you wish.

So long as the format of the 4 bytes in a 32-bit or 2 16-bit numbers is the same, many of the function for either type will in fact be identical. There is no need to distinguish between the two data types. However, in some systems, the organization of the bytes are different for the two data types.

MVP-FORTH was designed to handle double precision numbers as 32-bit numbers. The mnemonic '`D`' has been added as a prefix to those operations on double precision numbers. A prefix of '`2`' implies a quantity as used in some of the special numeric operators as `2*`, `2+`, `2/` etc. This is a very different meaning from '`D`'.

Only after MVP-FORTH was completed, was the origin of the '`2`' prefix on some of the stack operators discovered. That origin was the result of actually referring to 2 16-bit numbers as a pair of numbers. The intent was quantitative. There really were 2 16-bit values. Many mathematic constants can be easily described as the ratio of a pair of numbers. The arithmetic operations can be accomplished with some of the mixed arithmetic operators such as `*/` .

But to keep the thinking straight it is worth while keeping the distinction between 32-bit values and 2 16-bit values. To the extent that the actual machine functions may be the same as the FORTH function for each data type, you could use them interchangeably. However, you would be mixing data types in your thinking. Where the functions are the same, you can define the functions referring to one type as an alias of the other type. This is what has been done in MVP-FORTH.

The entering of double precision values is done by entering a decimal point somewhere with the number. No other symbol will be recognized by

MVP-FORTH. The decimal point causes the number interpreting routine to hold the value as a 32-bit value in 4 bytes. When such a value is stored on the data stack two stack locations are used. Remember that each stack location includes two bytes even if both are not used. To store a 32- bit number 4 bytes are needed.

The double precision value does not store with it the location of the decimal point. This is left for the programmer to manage. There are several methods of doing this. As with single precision 16-bit integers, the programmer must keep track of any possible decimal point. This is easily accomplished by understanding the use of a numerical value. It is best to think of a quantity in terms of the number of resolution elements.

If you are dealing with money, you have a choice. You can use the resolution element as a dollar. In today's markets that is usually sufficient. But if you wish you can make the resolution element cents or even mills for those who remember when that had any meaning. If you are dealing in distance you can make the element of distance inches, centimeters, miles, kilometers, microns, astronomical units, or anything else appropriate to the application.

Should you be working with money and wish to make the resolution element cents you would have to make sure that all values include two digits to the right of the decimal point. MVP-FORTH has a counter of the number of digits which were entered to the right of the decimal point. The decimal point locator is a user-variable `DPL`. After a number is interpreted, it is a simple matter to scale the value to the resolution limit.

```
: SCALE
   DPL @
   2 OVER  <  ABORT" Input Error "
   NEGATE 2+ ?DUP
   IF 0
      DO 10 1 M*/ LOOP
   THEN
;
```

There are many ways of doing your scaling. Perhaps this will serve as an example to get you started. Here some error protection is appropriate because the end user might not be as responsible as the programmer.

You can then explore the whole family of double precision number FORTH operators. They include the creation of double precision constants and variables: `DCONSTANT` and `DVARIABLE`.

There are a number of double precision operators such as: `D+`, `D-`, and mixed double and single precision operators such as `M*`, `M*/`, `M/`, `M/MOD`, and `U/MOD`. The purpose of this section is to call them to your attention. You can refer to ALL ABOUT FORTH for the functional definitions and examples of the use of each. Make up your own exercises.

Then there are a group of double precision stack operators such as: DDUP, DDROP, DSWAP, etc. You will find other double precision functions as: DMAX, DMIN, DMINUS, DNEGATE, DABS, etc. Also there are the output formatting functions for double precision numbers such as: D. and D.R.

Isn't it amazing how many words can be added to the vocabulary for such a simple additional capability as using double precision values? The scheme can be increased to handling n-precision numbers. Again a whole family of functions and definitions would be necessary. Unless you have special needs, there is no point in adding all of these functions. If you do want them you will also have to understand their use. To do that you might just as well write them on demand.

Should you go on to working with ratios, you have available a number of mixed operators such as */ and M*/. Some of these carry the intermediate value with increased precision. These are most convenient for maintaining accuracy in integer arithmetic.

You should be aware of a potential problem in using D! and D@ for 2! and 2@ in MVP-FORTH. In some data base definitions, the 2! and 2@ functions are used. The basic definitions of D! and D@ have been implemented in machine code. They work properly when used together. But the actual order in which the bytes are stored is not the same as you would expect with 2! and 2@. As implemented, the high order 16-bits are stored at the lower memory address rather than the higher memory address. The values are in the expected order when on the stack. It all depends upon what you expect. If you are doing much work in this area, you will probably want to redefine these two words before using them.

```
: 2! ( d addr --- )
   DUP ROT SWAP ! 2+ ! ;

: 2@ ( addr --- d )
   DUP @ SWAP 2+ @ ;
```

These two will also work together. D! and D@ would also work together properly if both were redefined in the same way. After redefining, any existing application program will function properly without change. However, if you have data which has been stored with one implementation and read with the other, you will obviously have problems.

If you recognize the existence of this potential problem, you can be sure that you understand the definitions you are using if you redefine them as a part of your application program.

This is an example of what may appear as a potential bug. In an effort to maintain the stability of MVP-FORTH the source code has not been changed. Efforts have been made to inform users of the potential problem. The current version of MVP-FORTH Version 1.x.03 has been stable since 1984. The stability has been judged to be a most important feature.

MVP-FORTH contains several number formatting routines. Some of them are based upon `D.R` which takes a double precision number and displays it right justified in a field of specified size.

In addition you can define your own functions to format the output any way you wish. The necessary primitives have names including the `#` symbol, 'sharp'.

Suppose you want to output dollars and cents without a dollar sign.

```
: .DOLLARS
   AMOUNT D@
   <# # # 46 HOLD #S #>
   TYPE ;
```

A double precision value is required for the input. The function of< # and #> is to begin and end the construction of an ASCII string from a double precision number. Each 'sharp' converts the next digit to the proper ASCII code and builds a string. In this example, `HOLD` is used to place the ASCII code for a decimal point into the string. `#S` completes the remainder of the value to digits and places them in the string. Finally, `TYPE` is used to display the assembled string.

Refer to ALL ABOUT FORTH for more examples of the number formatting functions.

To conclude, you should be clear in your thinking about the data types you are using for specific functions. FORTH does not do the data type checking for you. You are responsible. There is a clear distinction between double precision 32-bit numbers and 2 16-bit numbers.

CHAPTER X

STRUCTURES

MVP-FORTH includes three structure types: 1/ the conditional `IF ... ELSE ... THEN` ; 2/ a fixed range, repetitive loop, `DO ... LOOP` , with several variations and 3/ a repetitive structure, `BEGIN ... UNTIL` , also with a number of variations. These structures provide all of the conventional programming structures. You have already used two of these structures.

The conditional structure you have already seen in some of the examples. Its use is relatively straight forward once you appreciate its unique requirement. The last data given to the system before the beginning of the IF function must be a truth flag. That flag is consumed by the function. A value of zero indicates a false flag and any non-zero value is interpreted as

a true flag.

A selection of comparison functions are available in MVP-FORTH. These include = , , , 0 , 0, and 0= as well as some which are double precision operators. Their functions are clear enough. In all cases, any data being compared is consumed. To be sure of the function of a particular operator it is easy to interactively test the function.

```
2 3  >  .
2 3  <  .
```

It is often easier to verify the function rather than try to remember it or to look it up. The system is there ready to help you through interactive programming.

The function of the IF command is to set a system flag and on a zero value to branch. The primitive available for your use if you insist is 0BRANCH. The word IF must be compiled in a definition. The structure cannot be used outside of a definition. Thus you must write a colon definition to test or use the conditional IF ... ELSE ... THEN structure.

The function performed is to jump either to the high level FORTH code which follows the optional ELSE or to that following the THEN, if the flag is zero. Remember a zero flag is false. If the flag is non-zero, the high level FORTH code following the IF is executed up until the optional ELSE or through the THEN. If the ELSE construct is used, the system compiles an unconditional branch instruction with the primitive BRANCH to the code which follows the THEN. The primitive, BRANCH, is also available to you should you need it.

You have seen examples of the IF ... ELSE ... THEN structure in an earlier section. Perhaps you can contrive some to exercise it now.

```
: TEST
   IF ." The value is not zero."
   ELSE ." The value is zero. "
   THEN ;
```

You have an example of writing style to set off the structure. Begin each part of the structure on a new line. This way you will be able to spot missing matched pairs. If you do not match these structures, the definition will not compile and you will get the message: DEFINITION NOT FINISHED.

You can give the system a truth value and execute the function TEST. According to the value given, zero or non-zero, you will be given the appropriate message.

You might shorten this example, but it might be cryptic and make it harder to read. You be the judge.

```
: TEST1
```

```
." The value is "
IF ." not "
THEN ." zero. " ;
```

As you can see, there are many possible ways of programming a desired function. Each may well produce the desired result. There is no single correct way. The ultimate test is whether the function works correctly in all circumstances. In checking any new function it is good to examine particularly the boundary conditions. Perhaps you do not have a boundary condition as in this case.

And speaking of cases, a number of CASE functions have been written. It is interesting to find that the code actually compiled is usually the same as that compiled by a series of nested `IF ... ELSE` constructs.

```
: TEST2
   DUP IF ...
      ELSE IF ...
         ELSE IF ...
         THEN
      THEN
   THEN
;
```

You can build up any sort of system you wish. Perhaps you can find one of the many CASE definitions and test it. Do a `DUMP`, or if you have a decompile function available use that to see what the actual compiled FORTH code looks like.

In using special functions such as CASE, you are adding complexity to your vocabulary. You will probably find that it is easier to master the functions available in the kernel and completely understand the use of each than it is to learn and understand a larger number of utilities.

The choice of how you desire to work is left up to you. There is nothing in FORTH to force you to do many things in the manner already described. In MVP-FORTH, you are also given the source code and can make any personal changes you wish. A CASE function is included in the latest ALL ABOUT FORTH.

You saw the first part of the `DO ... LOOP` discussion earlier. It is repeated here for convenience.

First, the `DO` function requires two input values or parameters. They constitute a range. The first value is one more than the last iteration to be performed and the second is the first iteration to be performed. The current value of the index for the structure can be found with the MVP-FORTH word, `I` .

The function of LOOP is to terminate the iteration. If the value at I is equal to or greater than the terminating value the loop is terminated.

```
: TEST3
   10 0
   DO I .
   LOOP ;
```

This example will display the digits 0 through 9 and stop. The style is open. You are wasting a lot of space. But it tends to increase the clarity of the code you have written. You might wish to adopt some other style. In the example, the style is to begin a DO structure at the beginning of a line. Therefore, the values, i.e., the parameters, for the structure must be at the end of the preceding line.

For very short structures such as this one which include only two high level FORTH functions, you might prefer to place them together on a single line. There is nothing in FORTH which forces you to adopt any given style. That is left to you. If you adopt a readable style, you will be able to understand your program later. Few comments are needed. If you run everything together, you are responsible for your later difficulties in understanding and debugging your program. You also should remember that others may need to read your code.

LOOP increments the index by one on each iteration. There are two other variations of the terminating loop function. +LOOP increments the index by the value given the system immediately preceding the name.

```
: TEST4
   10 0
   DO I . 2
   +LOOP ;
```

For the sake of beginning the terminating line with +LOOP the incrementing value will always be on the end of the preceding line. Again this is a matter of style.

The increment is a simple signed arithmetic addition. Thus in signed integer arithmetic the limit is going to be 7FFF hexadecimal. The value 8000 hexadecimal is the lowest, not the highest, value and may cause trouble.

Occasionally, it is convenient to use addresses for the parameters of the LOOP function. You must then be careful to not cross the boundary for signed integer arithmetic. An alternate loop function, /LOOP, ('up-loop') does simple unsigned arithmetic. This can then be used to cross the boundary condition imposed by +LOOP.

DO ... LOOP structures can be nested indefinitely to the size of memory available. In nested structures, the value returned by I is always the value

for the current loop. The index for the preceding loop structure is returned with the function J . You might want to look up one other function which will return the index under special circumstances: I' . Nested DO ... LOOP structures must always be paired. Again using a convenient style will facilitate the pairing of all of the structures.

There are some occasions in which you may wish to leave the DO ... LOOP structure before the indexed iterations are completed. This can be done with the function LEAVE. Suppose you are executing a function and you realize that the wrong thing is happening. You would like to be able to terminate the structure.

```
: COPIES
   DO I . I I 100 + COPY
      ?TERMINAL IF LEAVE THEN
   LOOP FLUSH ;
```

This is the sort of utility you might write to copy a group of FORTH screens up 100 screens. The offset value of 100 in this case could be made equal to the offset from one disk drive to the next. The function would then copy the selected screens from one disk to the other.

One feature illustrated in this example is the use of I . 'dot' within the definition. The word I returns the value of the current iteration of the DO ... LOOP which you can then print. It is always nice to know how you are progressing in such functions.

The second is the use of ?TERMINAL to inspect the key board. If a key has been struck, the flag will change from zero to non-zero and you will force the loop to terminate at that point. The function ?TERMINAL has a potential problem among implementations of MVP-FORTH. The problem depends on whether the system has a type-ahead input buffer. Check the documentation for your system. In the example given here, there is no problem in any of the implementations of MVP-FORTH. This is just a word of caution.

The last structure is the BEGIN ... UNTIL structure. This general structure also has a number of variations.

```
: TEST5 0
   BEGIN 1+ DUP . DUP 9 =
   UNTIL DROP ;
```

The 0 provides the initial value for this example. That value is incremented and then displayed on each repetition. The value is then compared to 9 and a flag is set. If the flag is false the sequence will be repeated. When the flag is true the repetition is terminated. The example is contrived and would probably be better written as a DO ... LOOP structure. However, the program does exactly the same thing. This is another example of the multiple ways which can be used to write a program. There is no one correct way.

One limitation with a DO ... LOOP structure is that the sequence must run at least once. There are occasions where you would like to only conditionally do the loop at all. The DO ... LOOP could be executed within an IF ... THEN structure. Alternatively, you could use a BEGIN ... WHILE ... REPEAT variation of this structure.

```
: TEST6 0
   BEGIN DUP 10
      WHILE DUP . 1+
   REPEAT DROP ;
```

This is still another way to execute exactly the same function you saw earlier. The proper flag is set before the WHILE function and as long as the flag is true, non-zero, the sequence will repeat. When the flag is zero, the rest of the sequence is skipped and the structure execution terminated.

The function REPEAT compiles a related MVP-FORTH word AGAIN. It is the run time function of AGAIN which causes the unconditional repetition of the structure. If you have no means of leaving the structure you will be in an infinite loop. That is exactly what the run time FORTH program is doing.

Look up the definition of QUIT. You will see that it is an infinite loop. That will give you something more to think about.

We have covered the three basic structures available in MVP-FORTH: the conditional IF ... ELSE ... THEN and two repetitive structures, DO ... LOOP and BEGIN ... UNTIL. They provide much of the power of FORTH. If your system runs at all, you can be sure that these structures are functioning properly.

The function of the DO ... LOOP in MVP-FORTH has been made to conform with the requirements of the FORTH-79 Standard. This was done early on. But the code is unnecessarily complicated. In the MVP-FORTH PADS (Professional Applications Development System) implementation, the adherence to the Standard has been sacrificed for the sake of improved performance. It is highly unlikely that a user will notice the difference. For the sake of stability of the MVP-FORTH program, the implementation has not been changed.

The future holds the possibility of requiring yet another change in the implementation of the DO ... LOOP structure. In implementing a FORTH processor there are advantages to using a count down register rather than the limits and an iteration pointer. If such a hardware implementation becomes common, the function of loop will be changed. Hopefully, a new name will be found for the new function rather than changing the function of a previously used FORTH word.

CHAPTER XI

MISCELLANEOUS MVP-FORTH UTILITIES

There are a number of utilities available in MVP-FORTH. MVP-FORTH is in fact an application program. The nucleus or kernel of MVP-FORTH could be made much smaller but it would no longer serve the purpose of providing the user with a fully functional learning tool.

Some of the special features of MVP- FORTH are addressed in this section. They are arranged in no particular order.

Although commenting source code is often over done, it is good to include a sufficient number of comments while writing your programs. Comments are excluded from the compiler and are indicated by enclosing them within parentheses or preceding a line with a back slash. Either method works.

As already illustrated, the 0 line of each screen is by custom a comment line and is enclosed within parentheses. Often programmers include their initials and date on that line. If your system has a built in clock you might want to include the time on that line as well.

Older implementations also enclosed some primitive names within parentheses. When the names of these words were included within the parentheses of a comment, the comment was terminated. It is often terminated before the end of the intended comment. After considerable frustration trying to use good documentation and having the comment line terminated unexpectly, an alternate notation was adopted for indicating a primitive. The word is placed within angle brackets instead of parentheses. The angle brackets are the "less than" and "greater than" symbols. The technique provided a simple solution to the problem. The connotation of the word being a primitive is very obvious.

Examples of such run time primitive functions are the vectored routines of MVP- FORTH. They are placed within angle brackets.

Earlier you were told that you would be able to save the current operating object code. That can be done with the function `SAVE-FORTH`. Refer to the implementations in ALL ABOUT FORTH. The function is relatively long but provides an example of access to a number of system functions from MVP-FORTH.

The system calls access the underlying system disk drivers. These functions are essentially the same in CP/M-80, CP/M-86, and MS-DOS.

The implementation of SAVE-FORTH can be used as a model for access to most of the system functions.

This implementation uses the older file control blocks of DOS rather than the more recent handles. The implementation is simpler but does have its limitations. A new version could be added if you desired. It would provide an application for learning how to manage handles.

As implemented, the object code will be saved on drive 0 through the parameter of 0 in the system function 14 call with SYSCALL. The value of 0 is defined as a constant in the FORTH dictionary rather than being left for conversion. It is thus compiled as two bytes in the definition of SAVE-FORTH. Inspect the data field of SAVE-FORTH with DUMP.

Especially with hard disks being more common, it has become desirable to be able to select the drive on which to write the object code file. A simple patch to SAVE-FORTH, will make this possible.

```
0 CONSTANT DISK
```

DISK will be compiled as two bytes whose function is to return the current value of the constant. It has been initialized to 0. By finding the location of the code field address of the 0 following the value 0D hexadecimal (for system function 14) in a DUMP, you can replace the code field address of 0 with the code field address of DISK.

```
HEX ' 0 CFA . DECIMAL
```

This has been an aside. SAVE-FORTH runs the way it is. Study of the implementation will provide an example for a number of other programs to access disk operating system functions.

The name you select at the prompt can either be the same name as the file you are currently running or you can give it a new name. If you use the same name the previous directory entry will be deleted and then reused as the new name. At this point you could equally well give the name of the application as a name with the file type .COM, such as PROGRAM.COM.

Depending upon the size of the disk you are using, you will have more or less space to keep as many different applications as you wish. When doing development work it is often convenient to save successive object code images with different names. Then you have a series of backups for each level.

The use of the system to save object code images would be eliminated if we had systems fast enough to recompile an application to satisfy our impatience. Many common micro-processor systems will take several minutes to compile a moderate size program. In some systems it takes that long to compile the UTILITIES, SUPPLEMENTALS, EDITOR and ASSEMBLER, on top of the MVP-FORTH kernel. When the system can compile all of this in less than a second or two, there will be no need for

saving the object code module.

FLUSH is an early word in the history of FORTH. Its function is to write back to disk all screen buffers which have been marked with UPDATE. Whenever the text on a screen is modified, that screen buffer is marked with UPDATE. It appears that the name was thought inelegant and was replaced with SAVE-BUFFERS in FORTH-79. Very quickly most programmers gave SAVE-BUFFERS the alias FLUSH.

Perhaps you should examine the screen buffers. Each buffer contains 1024 bytes of data with two trailing bytes containing the value of 0, two leading bytes which contain the current screen number, and a flag for the UPDATE function. The contents of the screen buffer need not contain only text data. It may contain any series of 1024 bytes. Only if text material is present within the screen buffer can it be listed in a reasonable form with LIST.

The two bytes containing 0 in the trailer cause the screen to terminate the loading function. This prevents a run away system should you forget to terminate a colon definition with a semicolon. It prevents you from concatinating screens. As a basic rule, if you need more than one screen for a colon definition, you probably should rethink your implementation. In ALL ABOUT FORTH is a definition for --> which will allow you to develop bad habits.

The leading two bytes contain the absolute screen number. As long as you are on drive 0, it will be the same as the actual screen number. But if you should be using screens on a second disk drive, an offset is added to the screen number in labeling the buffer. The current offset is stored in a variable OFFSET. What could be simpler?

The highest order bit of the leading sixteen bits is set to mark the buffers for update. The function of UPDATE is to set that bit. When the buffer space is needed for another screen, it is first checked to see if it has been marked for update. If it has, it is written back to disk before it is used. If it is not, the buffer is simply written over.

You can inspect the header value easily.

```
75 BLOCK 2- @ .
```

This will show you the current block number in that buffer. Now try:

```
UPDATE 75 BLOCK 2- @ .
```

The memory used by FORTH is initialized to the number of buffers in the constant #BUFF. By changing the number in this constant, a different number of new buffers will be initialized with CHANGE. These buffers are a form of disk caching.

Perhaps you have made a series of modifications and then thought better of them. Maybe there will be other occasions where you would like not to

save the current changes to a screen back to disk. This can be avoided by the function of EMPTY-BUFFERS. All buffers are emptied. This command is also executed when the buffers are initialized. It is just as well that the word is a little difficult to type. It is destructive of information.

The use of the buffers in MVP-FORTH is cyclical in sequence. Both the run time functions of BLOCK and BUFFER will cycle through the available screen buffers. The variables USE and PREV are used in conjunction with these functions. USE is a variable providing the address to or from which data will be written from or to disk. The address currently in USE will be used by the disk access functions of the operating system. Should you change it from a pointer to a buffer, you will have to change it back when you again try to read screens to the buffers.

The beginning of the buffers is pointed to by a constant named FIRST. Below FIRST are several other areas. The return stack comes down from FIRST. It approaches the terminal input buffer which is pointed to by a user variable, TIB. The data stack, or parameter stack if you wish, starts from TIB and moves toward lower memory.

This area of memory can all be examined with the DUMP function. You will note however, that this is an active area of memory. The DUMP function uses the data stack and the return stack. Thus you cannot get a stable image of these areas.

The sixteen bit processor systems have a larger memory than the limited 64K available on most 8-bit processor systems. With the 80x86 family of processors, more than 16- bits of address are needed. With MVP-FORTH on these systems, you can read and write to any part of memory with long functions. These require that the address be given in 2 16-bit words with what is known as the segment address first. The common fetch and store functions are: !L , @L, C!L and C@L. You can also dump from any area with DUMPL. This function also requires 2 16-bit words for an address with the segment word first. You can use this function to examine the memory mapped display as well as the ROM memory in the system. MVP-FORTH program space is limited to a single segment but all memory segments are accessable. The DOS parameters in low memory can be examined and modified.

The dictionary search proceeds through the entries by a series of links. It is possible to restrict the search to specific segments of the vocabulary by placing entries in special areas known as vocabularies. The EDITOR and ASSEMBLER (hard copies of which are in the MVP-FORTH Documentation) are each in separate vocabularies. Unless one of these vocabularies is invoked by entering its name, no dictionary entries in the vocabularies will be found. This is one reason that you may not be able to use the line editor provided.

There are two flavors of vocabularies available in MVP-FORTH. In one all vocabularies chain directly to FORTH. That is, when a vocabulary is invoked, it will be searched first and then the FORTH vocabulary will be searched. If you were to define a second vocabulary within another vocabulary, the search would only search the current one and jump directly to the FORTH vocabulary. The other flavor allows the sequential searching of parent vocabularies all of the way back to FORTH. The latter is known as <VOCABULARYFIG>. The former is <VOCABULARY79>. The function of VOCABULARY is vectored through 'VOCABULARY.

```
VOCABULARY NEW IMMEDIATE
NEW DEFINITIONS
: NEW-TEST
   ." THIS IS THE NEW VOCABULARY." ;
FORTH DEFINITIONS
```

VOCABULARY is a defining word. It creates a dictionary entry for the new word following VOCABULARY as with any other defining word. All vocabularies are made IMMEDIATE. IMMEDIATE sets the second most significant bit in the word just defined. When this bit is set, the word will execute rather than compile when used within a colon definition. More on immediate words later.

Then to add definitions to the newly formed vocabulary, you must invoke the function of DEFINITIONS. With this function, all new definitions will be placed in the selected vocabulary until a new vocabulary is selected with DEFINITIONS. This is why you returned to FORTH DEFINITIONS.

The FORTH words CURRENT and CONTEXT are primitives used to select the proper vocabularies for searching and compiling.

You will now find that in the FORTH vocabulary, you will be unable to find NEW-TEST. However, by first invoking the vocabulary NEW, you can then execute NEW-TEST. The example provides a pattern for the use of vocabularies. It also gives you an idea of how EDITOR and ASSEMBLER work. You might examine the source screens for both of those vocabularies for further examples.

A name which has the flag set with IMMEDIATE will execute rather than compile during compilation. One can force the compilation of an immediate word with the function of [COMPILE].

```
: L ( --- )
   SCR @ LIST [COMPILE] EDITOR ;
```

This is a convenient utility. In the EDITOR is a word L which causes the current screen to be relisted. After loading a screen, you are left in the FORTH vocabulary. What you would like to do is list the same screen again and go back to the EDITOR. You could accomplish this with:

```
EDITOR L
```

But perhaps you forgot to write `EDITOR` and just used the word `L` . This mistake will give you an error message because `L` is not defined in the `FORTH` vocabulary. The variable `SCR` stores the address of the most recently accessed screen. Thus you can re`LIST` the screen easily enough. But `EDITOR` is an immediate word. Therefore, to force its compilation with your new `L` in the `FORTH` vocabulary you have to use `[COMPILE]`.

Whether you are compiling or executing is determined by a flag in the variable `STATE`. While you are in the compile state, you can temporarily leave and enter the execute state. In the execute state, the input stream is interpreted and executed. In this state, you can make calculations and return to the compile state. The flag in `STATE` is reset with the function `[` , and set with `]` .

This section has been rambling through a number of less commonly used MVP-FORTH functions. The discussion should spur you on to further experimentation.

CHAPTER XII

OPERATING SYSTEM INTERFACE

This is an advanced section. Most FORTH users will have no need to interface with an underlying operating system. Parhaps you are one of those who should skip this section.

The Language FORTH in its earliest days grew out of the need for an efficient operating system. With the expanding availability of computer memory, the system programmers have gotten carried away with the progressive "enhancements" to operating systems. Because systems also seem to be running faster, the cumbersome size and speed penalties are less obvious.

They are less obvious until you have a chance to see what an efficient operating system will do. FORTH programs have no need for the existing operating system. The only advantage is to have access to the succession of drivers. Aside from this feature, one whould do well to get rid of any underlying operating system.

But there are always a few who will insist on mixing their work. It is possible to access any system file with MVP-FORTH. It does require that

you understand what you are doing. You can use FORTH to strip the special codes in the text files produced by some word procesors.

You have to decide how to use the facilities of the computer system.

Well, in spite of this tirade, there are occsions when access to the operating system might be desirable. A few simple routines make this process relatively easy. Most implementations of MVP-FORTH have a function `SYSCALL`. This function requires as parameters the desired function of the operating system and the necessary parameter or a dummy parameter. The system function will be executed and any error message will be returned.

To make use of the system functions you will need to know what they are. That is beyond the scope of this section. You can look them up in your system documentation. Much of the discussion will apply to most systems. A few system functions take more than one parameter or return something unusual. These functions will not work with the implementated function of SYSCALL. However, you could model the necessary function on the implementation included and do anything you wish.

The most interesting use of the system is to read and write program and system files. All disk accesses using system calls require a file control block. This is a block of 32 bytes which contains file information. Seven bytes before the file control block and 3 bytes after it may also be used by some systems. Therefore you really need to allocate 42 bytes to a space for the file control block. The pointer to the file control block is actually seven bytes above its beginning. If you understand this you could make space in the FORTH dictionary for a file control block.

```
CREATE  fcb  44  ALLOT
: FCB  fcb  7  +  ;
```

You could make as many such file control blocks as you wish. Each would have to be given a separate name.

The next problem is to set up the necessary information in the block for the system to access. Why you do what you do you will have to learn from other materials. There are several FORTH functions and techniques which are well illustrate with the example below:

```
: MAKE-FCB   ( --- )
   14 0  SYSCALL DROP      \ Reset disk
   CR  CR                 ." Enter
             [d:]file name.typ ---"
   FCB 37 0 FILL  FCB 1+ 11 BLANK
   QUERY
   46 WORD  DUP 2+ C@ 58 =  \ ? [d:]
   IF DUP 1+ C@ 65 - 14 SWAP
      SYSCALL DROP          \ Select drive
      COUNT 2- SWAP 2+ SWAP
   ELSE COUNT
   THEN
```

```
8 MIN FCB 1+ SWAP CMOVE
BL WORD COUNT 3 MIN
FCB 9  SWAP CMOVE  CR  ;
```

This imposing looking definition is a step by step advance through the proces required. A prompt is used to indicate the imput form. The square brackets are commonly used in operating system documentation to indicate optional inclusion of the data indicated. Note that this does not allow for a path.

The current contents of the file control block are cleared and the name field is set to spaces. Then the FORTH function to collect data is issued. After that the first word returned is parsed up to the period which has the decimal code balue 46. The word is inspected to see if the second character is a colon, (58). If it is the first character is converted to the drive number and used to select the drive. Then the rest of the file name is placed in its field and finally the file type is placed in its field.

The paragraph describing the function is moderately long, nor does it include all of the necessary details. Presumably, by now you can read and understand FORTH.

There is no reason not to include the entire function as a single FORTH word. With this file control block you can perform any of the system functions on files. You can make the file, open the file, close the file, read and write randomly to the file as well as sequentially. But you have to decide what you want to do.

The file control block can also be used to select a file and send it out a modem or another computer on an RS-232 connection. A reciprocal function can be used to write a file from a modem or another computer.

This modification and the examples of the use of `SYSCALL` and `SAVE-FORTH` in ALL ABOUT FORTH should give you sufficient models with which to work. FORTH is a programming language. You as the programmer must decide what you want to analyze and then program for your specific application.

Perhaps this is a place to show how to deal with one of the most difficult problems in using multiple disk formats with one of the IBM-PC operating systems. IBM has several times changed their disk operating system. Somehow it is necessary to inform the disk operating system of the format of the disks currently in the system. A means for doing this automatically has avoided solution for some time. It was a much simpler matter to tell the FORTH COFIGURE routine what you know the format to be.

First you would have to cause the operating system to read the disk being used. For this you would have to leave FORTH and do a directory on the disk in each drive. You could change the disks in a drive without trouble as

long as you always used the same disk format. This works well for your standard mode of operation in which all disks will have the same format.

Only when you try to read a single sided disk formatted under DOS 1.0, for example, would you have trouble while using FORTH under DOS 2.1. A solution which can be executed from FORTH has been found.

It turned out that one of the advanced function calls in the disk operating system will do the job.

```
: RESET-DRIVES
   MAX-DRV  0
   DO  I DR-DEN  3  <
      IF  54 I 1+ SYSCALL  DROP
      THEN
   LOOP ;
```

Once the solution was found it was easy enough to implement. See your system documentation.

An interesting embellishment on the `CONFIGURE` function is to reset the existing floppy disks automatically. To do this you will have to understand the scheme used by IBM to code the current disk format. That information is in the zero track of the disk and can be accessed with from the data included in the 0 BLOCK of FORTH. The following code will then do the job. You do not even need to know what format a disk has.

```
: RECONFIGURE
   EMPTY-BUFFERS  RESET-DRIVES
   0 OFFSET !
   MAX-DRI  0
   DO  I DR-DEN 3 <
      IF  0 BLOCK 3 + C@  DUP 0=
        IF  0  ELSE DUP 3 =
          IF  1   ELSE  8 =
            IF  0  ELSE 0 BLOCK 512 =
              C@ 07 AND DUP 5 =
              IF  2  ELSE DUP 6 =
                IF  1  ELSE 7 =
                  IF  0  ELSE 1
                    ABORT" Non-IBM Disk"
        THEN THEN THEN THEN THEN THEN
        I 2* DEN + !  BPDRV OFFSET +!
       THEN
   LOOP ;
```

It is highly unlikely that you will use a single sided disk with nine sectors per track. Therefore, no provisions for this format has been made.

The high level FORTH code checks the appropriate bytes in the first and second sector of the zero track and according to the value found sets the necessary values for reconfiguring the IBM floppy drives.

CHAPTER XIII

EDITORS AND EDITOR

Editors are very personal. Most people find that the first editor they learned to use is the only type of editor to use.

But there are many different editors of a number of different types. The modern word processors are very powerful editors. The one being used to write this has more commands than MVP-FORTH. It has taken years to learn it and the revisions are more frustrating than helpful. The result is that an old version is being used.

In MVP-FORTH, all you need is to be able to write to the FORTH screens. Each screen has only 16 lines of 64 characters on each line. You do not need all of the power of a full blown word processor.

You have already been shown how to use the rudimentary editor which is included in the MVP-FORTH kernel.

```
75 LIST
75 CLEAR
0 PP ( THIS IS LINE 0 )
FLUSH
```

This sequence will allow you to develop programs in screens and test them with a minimum change in your previous habits. The initial translation of the EXPERT-2 System to MVP-FORTH was done with this rudimentary editor even though several types of FORTH editors were available. It includes the necessary and sufficient tools to progress very rapidly. Why would you want anything more complicated?

Well, there are a few things you would like to be able to do without having to type an entire line over. You would like to be able to find character strings and replace them. You would like to be able to insert and delete strings and whole lines and have the rest of the text expand and contract as necessary. Yes, something more than the rudimentary editor is available. But you will have to learn some more FORTH functions.

Charles Moore, who created FORTH because of the constraint of speed, created an editor. In his hands, screen editing can be done with amazing speed. If you are interested in speed, you might find it worth while taking a look at the capabilities of his editor.

Sam Daniels wrote an implementation of Charles Moore's editor and placed it in the public domain. It was published in FORTH

DIMENSIONS, Vol III No. 3. That editor has been translated to MVP-FORTH. It has been compiled into your implementation of FORTH. You have the source code both in hard copy with your MVP-FORTH system documentation, and on the MVP-FORTH screen's disk.

You will find many examples of FORTH functions in the source code. Do take some time to look at them. They will serve as models for developing a number of unrelated applications.

The first thing you will notice is that when you bring up your MVP-FORTH, none of the editor commands seem to work. Frustrating! The reason is trivial. Your run time MVP-FORTH dictionary is not burdened with the vocabulary necessary for the editor unless you need it.

Invoke the editor by executing `EDITOR`. Now suddenly you will find that the editor functions are available. Every time you `LOAD` a screen as you test a program during development, you will leave the `EDITOR` vocabulary. So you will have to invoke the editor again if you need it.

Well, what can you do with the `EDITOR` vocabulary? Once you have `LIST`ed a screen, you can relist that screen with a single character command, `L` . This command finds the number of the current screen in the MVP-FORTH variable, `SCR` , and executes `LIST`. You have seen this definition earlier.

```
: L   SCR @ LIST [COMPILE] EDITOR ;
```

You might even want to add this to your rudimentary editor by adding the definition to your MVP-FORTH vocabulary.

As you make changes you can always redisplay the revised screen with this command. The problem of displaying an image would be greatly simplified if all displays were memory mapped in your system. Unfortunately, this is not the case. In fact, the variety of monitors which connect to your system with an RS-232 line are each different and few are memory mapped as far as your system is concerned.

You will find that the functions of this `EDITOR` vocabulary will work on any system. Having learned the few commands you will use, you will be able to move from system to system with completely different hardware. No new learning curve for each.

When you list a screen while in the EDITOR, you will be shown one of the lines below the screen with a caret mark, in the line. Occasionally, you will get a message: NOT ON CURRENT SCREEN. This means that the pointer to the location of the cursor is larger than 1023, the number of characters possible on a screen. You can then tell the system the line you would like to place the cursor on.

```
0 T
```

This will type out the 0 line of the current screen. It will show you what you have written on the index line of screen 75, if that is where you are. You will also see a caret at the beginning of the line. That is where the cursor is currently located.

Using a simple command, `P` , for put, you can now replace the text which is on the selected line.

```
0 T P ( THIS IS LINE 0 )
0 PP ( THIS IS LINE 0 )
```

These two sequences do exactly the same thing. The name `PP` was adapted from `P` , but the character was used twice to distinguish it from the single character command. The function of `P` in the EDITOR requires that you first select the desired line with `T` , while `PP` requires that you tell the system the line number. It is easy enough to type a character twice once you have found it.

As a contrived example, insert something in the existing line.

```
F IS
I ON
L
```

You will find it easier to execute these sequences than to explain them. The name `F` is for find. The name `I` is for insert. The mnemonics are good and only a single key stroke is necessary. With the EDITOR functions, a single space is used for parsing. All subsequent spaces are significant.

Add some more text to your experimental screen. Use these commands to modify them. Perhaps you will want to erase something you have found. Try the name `E` for erase. You can also try `R` for replace. All good mnemonic, single stroke functions. No control characters to associate with any of them.

One other function is most useful, `TILL` . This erases everything from the current cursor position through the character string which follows `TILL` .

These functions are even smarter than at first appears. The `F` and `TILL` use the find buffer. Once something has been placed in a buffer, use of the function without adding anything to it will use the existing contents of the buffer. The `R` and `I` functions use the insert buffer. These functions will use the existing contents of that buffer, if no new contents are given.

You can use these commands in conjunction with one another and repeatedly.

Suppose you want to go through a screen and change all occurances of `DDUP` to simply `DUP`.

```
0 T F DDUP
R DUP
F
```

```
R
F
R
etc
```

When you get to the end of the screen you will see the find buffer displayed followed by NONE. Continuing to execute the F function will recycle through the same screen.

There are many other permutations and combinations of these commands which allow you to move rapidly through a screen. It is easier to say what you want to find than to move a cursor over the page. You can also be sure that you find every occurance of a particularly string on a screen. If you do not wish to make a change after you have placed the cursor in some location you can advance to the next location by repeating the F function.

Practice is the only way to learn these commands. There are only a few and each has a reasonably good mnemonic. You should master them quickly. While in the EDITOR, you can refresh you memory of the available commands with VLIST.

Using VLIST, you will see that the first word in the EDITOR's vocabulary is M . That is mnemonic for move. The source for the move is always on the current screen, i.e., the value stored in SCR, and the line is the one on which the cursor is currently located. You must tell the system which screen number ollowed by the line number above which you want for the destination line. The destination screen need not be different from the source screen, but you will have to tell the system that anyway.

```
0 T 75 0 M
```

Suppose you are still working on screen 75. The above sequence will make a copy of line 0 on line 1. All of the other lines will be pushed down one. What was on line 15 will be pushed off the screen and lost.

```
L
```

You can now see what you have done. If you are doing a number of moves you will automatically learn to follow it with an L. Perhaps that is a little cryptic for you. Once you move beyond the novice level, you will find that the commands are easy to use and very fast. Try VLIST again.

The word R, you have already learned. The word U is new. The function of U is to insert a new line below the current line. All lines are moved down one and what was on line 15 is lost.

```
0 T U : NOOP ;
U
U
L
```

The string buffer for the U command is the same as that for R and I. The function of U without a new string is to insert the contents of the

insert/replace/U buffer.

Try VLIST again. The next new word is S. This is a most convenient function. It will search as much of a disk as you wish for any string you wish. The search will always begin with the current screen. It must be told the number of the ending screen. Following the command you enter the string you wish to search for.

```
80 S LINE
```

All occurances of the word LINE will be displayed followed by the line number and the screen number. The power of this command is not limited to FORTH screens.

Perhaps you remember trying to find the system dictionary on one of the early screens of a system disk. Try it this way. Place a system disk in the the drive. Be sure you have a backup. You might do something you do not expect to do.

```
0 SCR !
10 S FORTH
```

You will see which block contains the directory entry FORTH. If there is an extension that will be found too. If you have an entry which has been erased you will see the location of that too. You can find and replace any string on any disk, including system disks. FORTH is powerful. It is also dangerous.

Referring back to VLIST, the next two words are primitives, so we will skip over them. The next potentially useful function is D. It is a combination of F and E. The combination is dangerous. The function will delete the string you enter without checking to be sure that is what you intended to do. You might find it more convenient to execute F . If you found what you expected you can use the E function.

Go back to your FORTH SCREENS disk.

```
75 LIST
```

Suppose you have finished that screen and want to go on to the next in order.

```
 N L
```

The word N, is mnemonic for next. The command increments to pointer to the current screen, SCR, by one. The function of L is to list the screen pointed to by SCR. You can now edit the next screen. You would have accomplished the same thing with:

```
76 LIST
```

To back up one screen you have available the function of B, mnemonic for back. The function of B is to decrement the value in the pointer SCR.

The last function crosses out the current line and moves all of the rest of

the lines up one. A blank line is added to the end of the screen.

```
8 T X
```

The line which is crossed out is placed into the insert/replace/U buffer. You can get it back! You can move the cursor to another line and insert the contents of the buffer under it with U.

```
6 T U
```

Having crossed out line 8, you then inserted it below line 6, i.e. on line 7.

There is no end to the possible combinations you can use with these simple mnemonic commands. With the exception of TILL, all of the useful commands are simple single key words. You can move around the screen with great speed.

Along with the source code for the EDITOR vocabulary, three useful words are added to the FORTH vocabulary.

The function of WIPE is to erase the current screen.

```
75 LIST
WIPE
75 LIST
75 LIST
75 CLEAR
75 LIST
```

Each of the above groups of commands does exactly the same thing.

The function of LINE is to leave the address of the beginning of the specified line number for various functions in the EDITOR. You could examine its function with your screen 75.

```
75 BLOCK  .
0  LINE   .
```

You will recall that BLOCK leaves the memory address of the specified block which you displayed. In the second example you used LINE to leave the memory address of the beginning of the specified line and displayed that. You will see that the two values are the same.

The last MVP-FORTH function added with EDITOR is MATCH and its primitive <MATCH>. It would be better to implement these functions in assembly code for the specific processors. But then the function would not be portable among various systems, so we sacrifice a small amount of speed.

The function of MATCH requires an address, the length of the field in memory which you wish to search for a match, and the address and length of the string being searched for. That is perhaps a little confusing. You need two pairs of addresses and counts. The first has to do with the range to be searched. The second has to do with the location of the string being looked for.

```
75 LIST
8  PP  THIS IS A STRING
1 TEXT STR
75 BLOCK 1024 PAD COUNT MATCH
```

You will investigate the function of TEXT and PAD shortly. The sequence of instructions will find the location of STR as an offset in the buffer for screen 75. After executing the above series of commands, prove the statement with:

```
EDITOR
R# !
L
DROP
```

You placed the cursor where you found the matching string and then redisplayed the screen with L. Under the new screen display you should see the cursor at the appropriate point ready for other functions. You will have to DROP the flag which MATCH leaves under the offset value.

The function of TEXT is to use the indicated value as a terminator for the input of a string of characters to a temporary location pointed to by PAD. You can test this with a simple example:

```
1 TEXT THIS IS A TEST
PAD COUNT TYPE
PAD 10 DUMP
```

As a result of the last sequence, you will see that the first byte beginning at the location pointed to by PAD is a length byte. The function of COUNT is to fetch the length and advance the address by 1. The order needs to be reversed for the convenience of TYPE. The function of TYPE is to display the text beginning at a specified address and of a specified length.

As with many of these examples, you should be working with ALL ABOUT FORTH and your MVP-FORTH Users's Manual. Use the examples there too. The purpose of this discussion is to make learning about MVP-FORTH a little easier than just reading the glossary, source code and documentation.

There are a number of other editors available, many of which are in the public domain. You can play with them as you have time. Each is designed to meet the needs of the author. Perhaps one of the other editors will meet your needs better. However, you are encouraged to give this EDITOR a chance. It was designed for speed and convenience of an experienced user. When you become experienced with its use, you may agree. Chuck Moore still uses his.

CHAPTER XIV

ASSEMBLERS

There is no way to write a general guide to all of the assemblers necessary for all of the different processors running MVP-FORTH. If you want to use an assembler, you will have to refer to appropriate materials for the processor in the system you have.

The several MVP-FORTH assemblers use the mnemonics which are standard for the system upon which the particular implementation is designed to run. How can one say anything in general? Well there are some things in common.

To define a word in the FORTH dictionary which will run the machine code generated by an assembler, a special header structure is necessary.

The problem is to create in the dictionary a header structure in which the code field contents points to machine language beginning in the data field. That machine code must end with a jump to a location known as NEXT in the object code of MVP-FORTH. In MVP-FORTH, that address is contained in a constant `NEXT`.

In 8080 machine code you could create a machine code routine which does nothing as follows:

```
HEX
CREATE NOOP   HERE DUP 2- !
C3 C, NEXT ,
DECIMAL
```

Between the `!` and `C3`, you could 'comma' and 'C-comma'in any machine code sequence you wish. You would be working back before the days of assemblers. The exercise is a good one to understand what you are doing but really not all that practical.

The source code for the assembler in your implementation of MVP-FORTH is available for your use in the MVP-FORTH User's Manual and on the FORTH screens disk provided. You will need to refer to that specific documentation. There are, however, some general comments. Included with the floating point application is a more complete ASSEMBLER.

The `ASSEMBLER` is a vocabulary in your implementation. To use it, you must call upon the `ASSEMBLER`. Remember that all vocabularies are flagged as `IMMEDIATE`. This means that you will have to make a special effort to compile it within a colon definition.

`ENTERCODE` is an MVP-FORTH primitive whose function accomplishes that and provides a check value for one level of security. This function can

be used in two places. There are two common ways to start assembly code: 1/ terminate a colon definition and continue with a machine code sequence which is the function of ;CODE , and 2/ create a new definition which is exclusively machine code and is the function of CODE . Both of these possibilities call upon the function of ENTERCODE.

```
: NOOP ;CODE  NEXT JMP  END-CODE
   CODE NOOP  NEXT JMP  END-CODE
```

The above sequences have the same function. They each create a header structure in the FORTH dictionary. Both functions end up doing nothing. Examine the source code for your assembler.

In MVP-FORTH, assembler functions defined through the use of ENTERCODE must always terminate with END-CODE. This is necessary to execute the security included in the defining words. Only if the security is met, will the smudge bit in the newly defined word be toggled off.

In order to place the assembler words in the ASSEMBLER vocabulary, you need to use;

```
ASSEMBLER DEFINITIONS
```

The structure of any assembler in FORTH takes advantage of a family of defining words. In all assemblers, an op-code may take none, one or more operands. The various op-codes fall into groups or types. A defining word for each type is first written. The number of types varies among assemblers. With these defining words, the specific assembler mnemonics are defined as words in the FORTH ASSEMBLER vocabulary.

You will need to examine your MVP-FORTH User's Manual to understand exactly what is being done. You will note that extensive manipulation of the values given the system is done by most of the defining words. As usual in FORTH, the system needs to know the values before receiving the actual command. All Forth assemblers require the parameters before the assembler function, rather than after as in conventional assemblers.

About the only way to master the use of a FORTH assembler is to examine code written with one. A number of the implementations in ALL ABOUT FORTH are given using an 80x86 FORTH assembler.

You will find one problem in the examples in ALL ABOUT FORTH. They were interpreted by a FORTH cross compiler which could do forward referencing. That is, you could refer to an address with a label and only later define the label. The need to do this is handled in a different way in most FORTH assemblers.

The need for forward references in assemble code has to do with a variety of structures. These structures include conditional forward jumps and loops. These structures resemble the conventional ones in FORTH and are given the same name but with different functions in the ASSEMBLER

vocabulary.

By examining those definitions in your assembler source code you can see how they operate. An alternate way is to really understand the machine code for your system and keep track of values yourself. In some ways this is the quicker way to a solution for one already experienced with the various conditional jumps.

The execution of assembler mnemonics in the ASSEMBLER vocabulary actually compiles bytes into the FORTH dictionary with 'comma' and 'C-comma' in the defining words. At all times the rest of the FORTH vocabulary is available. Of course you will have to exclude those structures whose functions are different in the ASSEMBLER vocabulary.

Where you want a label for a forward reference, you can save the location with `HERE` and give the system a dummy value. If the value goes into an address following a conditional jump, you will have to increment the value found by `HERE`. When you get to the location to which you want to do the conditional jump, you can get its location with `HERE` and store that address in the previous address.

```
...  HERE 1+  FFFF JNZ  ...
...  HERE SWAP !  ...
```

All of this requires understanding what you are doing. This is not designed as a tutorial in assembly language. Rather, you might find some guidance in techniques to explore.

Try taking a tutorial for the assembler language for your machine. Start at the beginning and implement the examples in FORTH. When you can make the transition, you will have mastered your machine!

Would that there were an easier method to master the use of machine language.